TX Text Control

ActiveX Programmer's Guide

Version 7.0

TX Text Control 7.0

Contents

What's New

What's New in Version 7.0 since Version 6.0

This chapter provides a general list of features that have been added or changed since Text Control version 6.0.

New Features

Text Control supports headers and footers. Several properties, methods and events have been added for this feature. See chapter "*Overviews - Headers and Footers*" for more information about these properties and methods and how headers and footers can be used and programmed.

Text Control supports several special types of marked text fields, like source and destination fields for hypertext links or fields that display the current page number. The new **FieldType** and **FieldTypeData** properties set the type for a marked text field and additional data depending on this type. See chapter "*Overviews - Marked Text Fields - Special Types of Marked Text Fields*" for more information about these features.

Text Control offers an additional page view that centers the document in the control's window and displays three-dimensional pages with shadows. This mode can be set with the **ViewMode** property. See chapter "*Overviews - Text Formatting and Views*" for more information.

Two new attributes that can be set with the **LoadSaveAttribute** property have been implemented. These can be used to specify an absolute path (**txAbsPath**) and a base path (**txBasePath**), to find files and other resources integrated in documents.

The new **ResetContents** method can be used to delete the entire contents of a Text Control.

The new **InputPosFromPoint** method can be used to calculate a text positon belonging to a certain geometric position.

The new **ObjectItem** property can be used to get a reference to an object's property and method interface.

Changes and Extensions

The Text Control text format has been changed to support headers and footers and special types of marked text fields. The text format is fully compatible to prior formats. Furthermore all prior formats can be loaded. The new format version number is 700. More information about how headers and footers are integrated, can be found in Appendix A of the DLL reference manual.

The **CurrentInputPosition** property can be used to set a new text input position.

New and Extended Properties, Methods and Events

Property/Method/Event	Description
CurrentInputPosition Property	Now supports the setting of a new input position.
FieldAtInputPos Property	Returns the field identifier of the field containing the current input position.
FieldGoto Method	Sets the current input position to the beginning of a marked text field.
FieldLinkClicked Event	Occurs when a marked text field is clicked that represents the source of a hypertext link.
FieldNext Method	Additionally supports the special field types
FieldType Property	Sets or returns the type of a marked text field.
FieldTypeData Property	Sets or returns the data belonging to a marked text field of a special type.

HeaderFooter Property	Determines which headers and/or footers the document contains.
HeaderFooterActivate Method	Activates a certain header or footer.
HeaderFooterActivated Event	Occurs when a header or footer has been activated.
HeaderFooterDeactivated Event	Occurs when a header or footer has been deactivated.
HeaderFooterPosition Property	Specifies a header's or footer's position.
HeaderFooterSelect Method	Selects a certain header or footer to use a Text Control property for the header or footer instead for the main text.
HeaderFooterStyle Property	Determines style settings for headers and footers.
InputPosFromPoint Method	Calculates a text position belonging to a certain geometric position.
LoadSaveAttribute Property	Supports the additional attributes **txBasePath, txAbsPath, txEnableLinks**, **txEnableHighlights** and **txEnableTopics**.
ObjectItem Property	Gets a reference to an embedded OLE object.
ResetContents Method	Deletes the complete contents of a Text Control.
ViewMode Property	Supports an additional page view that centers the document in the control and displays three-dimensional pages with shadows.

What's New in Version 7.0 since Version 5.2

This chapter provides a general list of features that have been added or changed since Text Control version 5.2.

New Features

The 32 bit version of Text Control has been extended to support Unicode, the character set for all languages. When using the DLL interface see the new chapter 1.14 "*ANSI and Unicode*" in the DLL reference manual for more information and a complete list of the extended messages and functions and how to use them. Unicode support is available on Windows NT and Windows 95/98.

The 32 bit version of Text Control now supports Far Eastern writing systems (Input Method Editors) and can process double-byte character sets. Internal dialog boxes and user messages are available in Japanese. This also is supported on Windows NT and Windows 95/98.

The **Load** and **Save** methods support loading and storing Unicode text either as text only or integrated in the Text Control's text format.

The new methods **SaveToMemory** and **LoadFromMemory** can be used to copy or load formatted or unformatted text to or from a byte array.

The new **Find** Method can be used to search for a certain string in the Text Control's text contents without using the system-defined dialog boxes.

The new property **CurrentInputPosition** returns page, line and column number of the current input position.

The new **TableNext** method can be used to enumerate all tables a Text Control contains.

The new property **TableCellAttribute** sets attributes of table cells like border widths, text distances and background color.

The new properties **TableCellStart** and **TableCellLength** can be used to get the start character index and the length of the text in a table cell.

The new **ResourceFile** property returns or sets the file name of a recource library which Text Control loads when resources are needed. This property can be used to display information strings and dialog boxes in other than the buil-in languages. See the new chapter "*Overviews - Resources*" for more information how to build a resource library.

A new tabulator type has been implemented. This type acts like a right-aligned tabulator but its position is always the rightmost text position. This tabulator type can only be set with the **TabType** Property.

Changes and Extensions

The Text Control's file format has been extended to support Unicode. See the "*DLL Reference Manual*" for more information.

The **DataFormat** property has been renamed to **DataTextFormat** to offer compatibility with Visual Basic 6.0.

In table cells with a single decimal tabulator, text is automatically formatted. It is not necessary to type a tabulator character.

New and Extended Properties, Methods and Events

Property/Method/Event	Description
CurrentInputPosition Property	Returns page, line and column number of the current text input position.
Load/Save Methods	Support the new format identifiers: 6 - Text only in Unicode format (Windows compatible). 7 - Text only in Unicode format (Text Control compatible). 8 - Internal Text Control format. Text is stored in Unicode.
Find Method	Searches the text in a Text Control for a given string.

LoadFromMemory Method	Loads text data in a certain format from a byte array.
ResourceFile Property	Returns or sets the file name of a resource library.
SaveToMemory Method	Stores text data in a certain format in a byte array.
TableCellAttribute Property	Sets attributes of one or more table cells.
TableCellLength Property	Returns the number of characters in a table cell.
TableCellStart Property	Returns the character index of the first character in a table cell.
TableNext Method	Can be used to enumerate all tables of a Text Control.
TabType Property	Supports the new tabulator type: 5 - Right tab at the right most text position. For this type any position set with the **TabPos** property is ignored.
Button Bar:	
Appearance Property	Returns or sets the painting style of a Button Bar.
ResourceFile Property	Returns or sets the file name of a resource library.
Style Property	Returns or sets the painting style of a Button Bar's buttons.
Status Bar and Ruler:	
ResourceFile Property	Returns or sets the file name of a resource library.

Introduction

Welcome to TX Text Control, the text processor in a single ActiveX control. Using Text Control, you can create all kinds of text-based applications with the ease of programming that is characteristic of Visual Basic and with highly sophisticated formatting and display capabilities which are normally the exlusive domain of large word processing packages.

System Requirements

The Text Control ActiveX control requires the following minimum configuration:

- Windows 95/98, Windows NT 4.0 or Windows 2000.
- Microsoft Visual Basic, Borland Delphi, Microsoft Visual C++ or one of many other development platforms which support ActiveX controls.

How this Manual is Organized

- Part 1 of this manual, *"Visual Basic User's Guide"*, describes how to use Text Control with Visual Basic 4 or higher.
- Part 2, *"Delphi User's Guide"*, shows you how to install and use Text Control with Delphi 2 or higher.
- Part 3, *"Other Languages"*, contains tips for using Text Control with languages other than Visual Basic and Delphi.
- Part 4, *"Reference"*, starts with several articles, giving you an overview how Text Control's properties, methods and events work together followed by a list of all Text Control's properties, methods and events.
- Appendix A describes Text Control's keyboard and mouse interface.

Distributing your Applications

The table below shows all the files necessary for Text Control to operate properly. You must ensure that these files exist on your client's machine and they are the correct version. If your client's machine has older versions of these files, you should update them.

1	TX4OLE.OCX
2	TX32.DLL TXTLS32.DLL WNDTLS32.DLL TXOBJ32.DLL IC32.DLL IC32.INI TX_BMP32.FLT TX_TIF32.FLT TX_WMF32.FLT TX_RTF32.DLL TX_HTM32.DLL TX_WORD.DLL
3	MFC40.DLL MSVCRT40.DLL
4	TX_GIF32.FLT

The first file (group 1) is the Text Control ActiveX server containing the ActiveX controls. These controls must be registered in the registration database on your client's machine.

The files listed in the second group are the additional Text Control DLL files. They must be installed in the same directory as the ActiveX server. You must always install all of them.

You should also verify that the Microsoft foundation class library files (group 3) are installed on your client's computer. These files must be

installed in the Windows system directory. Please refer to Microsoft's redistribution policy if you need to redistribute them.

The last file (group 4) is a filter to use the GIF image format with Text Control. Unisys Corporation holds patent rights to the LZW technology used in this filter. If a customer wants to use the GIF file format, he is required to obtain a license from Unisys and send a copy of the license agreement to The Imaging Source Europe GmbH. We will then send him the GIF filter free of charge.

Visual Basic User's Guide

Creating a Simple Word Processor

This chapter shows you how to create a small word processor from scratch with just a few lines of code. It will be able to load and save files, use the clipboard, and will have dialog boxes for character and paragraph formatting, a ruler, a status bar and full keyboard and mouse interface. The source code for this example is contained in the *Simple* sample source directory.

Creating the Project

Assuming that you have already run the Text Control installation program and started Visual Basic, the next step is to create a project for the text processor. To do this begin by selecting the *New Project* command from the file menu. Then use the *Tools / Custom Controls...* command to include the file 'tx4ole.ocx' into the new project. You will see four additional icons appear at the bottom of the toolbox, representing the Text Control and its Status Bar, Button Bar and Ruler:

 The Text Control Icon

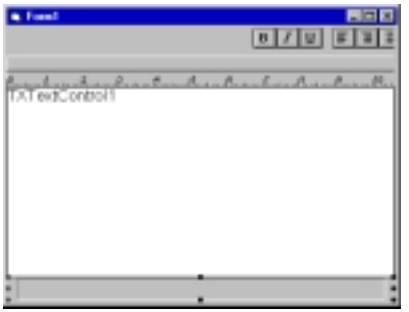
 The Status Bar Icon

 The Button Bar Icon

 The Ruler Icon

Creating the Controls

The next step is to put these four controls onto a form and connect them. Click on the Text Control icon and draw it on the form. In the same way, create a Ruler and a Button Bar on top of the Text Control, and a Status Bar below it. Your form should now look like the diagram on the right:



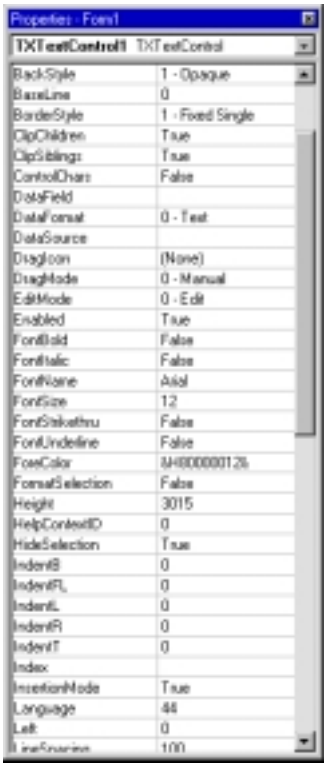
Connecting the Controls

Add the following code to the form's **Load** event procedure:

```
Private Sub Form_Load()
    TXTextControl1.ButtonBarHandle = TXButtonBar1.hWnd
    TXTextControl1.RulerHandle = TXRuler1.hWnd
    TXTextControl1.StatusBarHandle = TXStatusBar1.hWnd
End Sub
```

Running the Program

The text processor is not yet finished, but we can make a first attempt at running it to see what it can do. Click the 'Start' button. You can type in some text, select it with the mouse, copy it to the clipboard (use the <CTRL>+<C> and <CTRL>+<V> keys as long as there is no menu), select a different font, set tabs and do lots of other things. All of these features have been built into the Text Control and can be used with almost no programming effort.



You will have noticed, however, that some features are still missing. For instance, if you resize the main window, the controls keep their old sizes. There is no menu, and there are no scrollbars either. We will fix this in the coming chapters.

Adding Scrollbars

To add scrollbars, click on the Text Control window to have its property list displayed. Click on the **Scrollbars** property and select *3 - Both*. Select the **PageWidth** property and enter 12000, which is about the width of a letter in twips, the currently selected measurement. Set **PageHeight** to 15000 for now.

Resizing the Controls

Two steps are involved in making the controls resize properly when the main window is resized.

- Set the **Align** property to *1 - Align Top* for the Button Bar, the Ruler and the Text Control. Set it to *2 - Align Bottom* for the Status Bar. This will adjust everything except the height of the Text Control.
- Open the code window for the form which contains the Text Control. In the combo boxes on top of the code window, select 'Form' in the 'Object:' box and 'Resize' in the 'Proc:' box. The code window should show an empty procedure for the **Resize** event:

```
Private Sub Form_Resize ()
End Sub
```

Extend it as follows:

```
Private Sub Form_Resize ()
    TXTextControl1.Height = ScaleHeight - TXRuler1.Height _
    - TXStatusBar1.Height - TXButtonBar1.Height
End Sub
```

This line of code will cause the Text Control's height to be adjusted every time the size of the form is altered. (The ' _' character is used to extend one logical line of code to two or more physical lines).

Adding a Menu

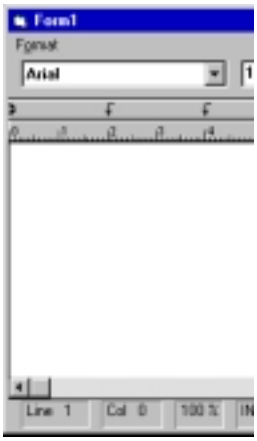
In this section, you will add a menu to the text processor to enable you to call the Text Control's built-in dialog boxes.

Use the Visual Basic Menu Editor to create a *Format* menu with the items *Character...* and *Paragraph...*.

Name the items 'mnuFomat_Character' and 'mnuFormat_Paragraph'. (Please refer to the Visual Basic documentation if you need help with creating menus).

Add the following code to the Click procedures of the menu items:

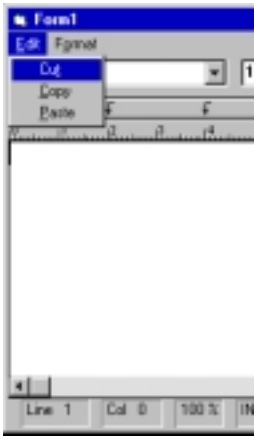
```
Private Sub mnuFormat_Character_Click()
   TXTextControl1.FontDialog
End Sub

Private Sub mnuFormat_Paragraph_Click()
   TXTextControl1.ParagraphDialog
End Sub
```



Start the program again. You should be able to use the menu items to call the Font and Paragraph dialog boxes.

Now for the *Edit* menu. Again use the Menu Design Window and create an *Edit* menu containing items for *Cut*, *Copy*, and *Paste*. The code for these menu items is:

```
Private Sub mnuEdit_Cut_Click()
   TXTextControl1.Clip 1
End Sub

Private Sub mnuEdit_Copy_Click()
   TXTextControl1.Clip 2

End Sub

Private Sub mnuEdit_Paste_Click()
   TXTextControl1.Clip 3
End Sub
```



Having added these menu items, you can exchange formatted text with other word processors via the clipboard.

Finally, we shall add one last menu. Create a *File* menu including the items *Load...* and *Save As...*. Place a common dialog box icon on the form and enter the following code, which will call the

common dialog box to get a file name from the user, and will then load respectively save the selected file:

```
Private Sub mnuFile_Load_Click()
   On Error Resume Next

   ' Create an "Open File" dialog box
   CommonDialog1.Filter = "TX Demo (*.tx)|*.tx"
   CommonDialog1.DialogTitle = "Open"
   CommonDialog1.Flags = cdlOFNFileMustExist Or _
      cdlOFNHideReadOnly
   CommonDialog1.CancelError = True
   CommonDialog1.ShowOpen
   If Err Then Exit Sub

   ' Pass the filename to the text control
   TXTextControl1.Load CommonDialog1.filename, 0
End Sub

Private Sub mnuFile_SaveAs_Click()
   On Error Resume Next

   ' Create a "Save File" dialog box
   CommonDialog1.Filter = "TX Demo (*.tx)|*.tx"
   CommonDialog1.DialogTitle = "Save As"
   CommonDialog1.Flags = cdlOFNOverwritePrompt Or _
      cdlOFNHideReadOnly
   CommonDialog1.CancelError = True
   CommonDialog1.ShowSave
   If Err Then Exit Sub

   ' Open the selected file
   TXTextControl1.Save CommonDialog1.filename, 0
End Sub
```

What Comes Next

It goes, of course, without saying that Text Control has many more features than those included in our little demo program. It is up to you now to include zoom, images, tables, OLE objects, paragraph frames and whatever else makes up a full-blown word processor. If you need some hints about how to integrate special features, have a look at the source code of the other sample programs or post a message in the Text Control support forum at http://www.textcontrol.com.

Text Control Programming

This chapter is a guide to programming Text Control and its tools, explaining the parts which have been omitted from the *Creating a Simple Word Processor* example.

Working with Files

Text Control uses 5 different file formats:

- Its own native format, which you would normally use to store data in document files.
- The Rich Text Format (RTF), which can be used to exchange formatted text with other applications.
- HTML
- Microsoft Word format
- Unformatted text in ANSI or Unicode format.

An example of how to use the native file format has already been presented in the previous chapter. Using RTF, HTML, Word or unformatted text is just as simple: All you have to do is specify the format you want to use as a parameter of the **Load** or **Save** method.

Using RTF, HTML, Word and unformatted text you can only read or write the contents of a single Text Control from or to a file. Using the native file format, however, you can write a file header prior to saving the Text Control data, or even write the contents of several Text Controls to one file.

The *Forms1* sample program, which is described in the next but one section, shows you how to write the contents of multiple Text Controls to a single file. The *MDIDemo* sample shows you how to write a file header prior to the Text Control's data and how to use RTF, HTML, Word and unformatted text.

Printing

Visual Basic provides two techniques for sending information to the printer. The first one is to use the **PrintForm** method, the second is to use the printer object. Both methods have their drawbacks: **PrintForm** works with screen resolution only, which would result in very poor print quality. The printer object, on the other hand, provides the best print quality, but requires a lot of coding. Text Control uses the second method to achieve the best result, but without a 'lot of coding'.

The following example sends the contents of a Text Control, which can be several pages long, to the default printer:

```
Sub mnuFile_Print_Click ()
   Dim wPages As Integer, No As Integer
   wPages = TXTextControl1.CurrentPages
   For No = 1 To wPages
      Printer.Print
      TXTextControl1.PrintDevice = Printer.hDC
      TXTextControl1.PrintPage No
      Printer.NewPage
   Next No
   Printer.EndDoc
End Sub
```

After storing the number of pages in a local variable called *wPages*, the printer object is initialized with the **Printer.Print** statement, The **For .. Next** loop runs from 1 to *wPages* to print all of the pages. Inside the loop there are three more lines of code which print a single page:

1. The device context handle of the printer object is assigned to Text Control's **PrintDevice** property. Without this step, a device context which is compatible to the screen device would be used, resulting again in poor print quality.
2. The number of the page to be printed is passed as a parameter to the **PrintPage** method. This will also start the printing process.
3. The printer object's **NewPage** method is invoked to advance to the next page.

Everything else, like calculating the line and page breaks, is done internally by Text Control. The formatting is based on the values of two groups of properties:

- **PageHeight** and **PageWidth** determine the dimensions of the printed page.
- **PageMarginB**, **PageMarginL**, **PageMarginR** and **PageMarginT** determine the print margins.

These properties are normally set in a page setup dialog box.

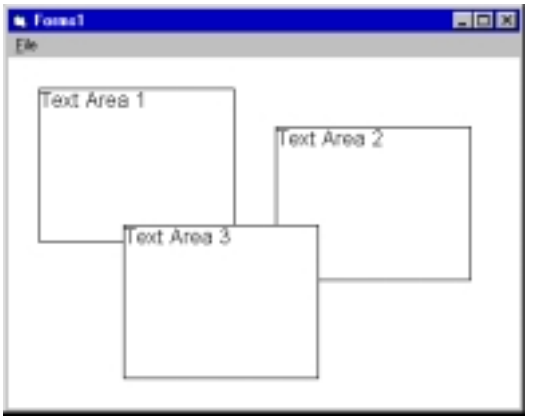
Using Multiple Controls

This chapter shows how to use Text Control in programs which have several text fields placed on a single page. Think of a program to print labels, to fill out forms, or to mask data entry. The *Forms1* sample program, which can be found in the *samples* subdirectory provides the basic functionality for applications of this kind.

Running the Sample Program

Initially, when the program is started, the main window contains one framed Text Control where text can be entered. The rest of the window is empty.

What you can do with the program is:



- Move the Text Control by pressing the ALT key and dragging the window with the mouse.
- Resize the Text Control by pressing the ALT key and dragging the window borders with the mouse.
- Create additional controls by clicking on an empty part of the main window.
- Save, load or print.

To keep things simple, there are no scrollbars in the main window and no menu items except the ones listed above. Scrollbars, zoom and a few other features will however be added in the next chapter.

How it Works

The *Forms1* sample uses a control array for the text fields. The first Text Control, the one which you see when you start the program, is placed on the form at design time. More controls are created when you click on an empty area of the form. These controls are created dynamically with the Visual Basic **Load** function when a **MouseDown** event occurs on the form:

```
Private Sub Form_MouseDown(Button As Integer, _
     Shift As Integer, X As Single, Y As Single)
  MaxID = MaxID + 1
  Load TXTextControl1(MaxID)
  TXTextControl1(MaxID).Move X, Y
  TXTextControl1(MaxID).Visible = True
  TXTextControl1(MaxID).ZOrder
End Sub
```

Clicking on an existing text field brings it to the front. This is done by changing the Z order when a Click Event has occured:

```
Private Sub TXTextControl1_Click(Index As Integer)
  TXTextControl1(Index).ZOrder
End Sub
```

The global variable *MaxID* counts the total number of controls; It is initialized to a value of 1 when the form is loaded.

Moving and resizing the controls is done by Text Control itself. To enable these functions, the **SizeMode** property must be set to *3 - Move and Sizeable*.

Saving the Controls

Saving a document which has been created with this program necessitates storing not only the data contained in the Text Controls, but also the number and the positions of the controls. In addition, a format identifier should be stored to enable the load routine of the program to determine if it can process a file which it is about to load. The code on the following page shows you how to save the document.

```
Private Sub mnuFile_SaveAs_Click()
   On Error Resume Next

   Dim i As Integer, FileID As Long
   Dim xPos As Single, yPos As Single
   Dim xSize As Single, ySize As Single

   ' Create a "Save File" dialog box
   CommonDialog1.Filter = "TX Form Demo (*.txf)|*.txf"
   CommonDialog1.DialogTitle = "Save As"
   CommonDialog1.Flags = cdlOFNOverwritePrompt Or _
      OFNHideReadOnly
   CommonDialog1.CancelError = True
   CommonDialog1.ShowSave
   If Err Then Exit Sub

   ' Open the file
   Open CommonDialog1.filename For Binary As #1
   If Err Then
      MsgBox "Can't open file: " + CommonDialog1.filename
      Exit Sub
   End If

   ' Write file header consisting of file format ID
   ' and number of controls
   FileID = FILE_ID
   Put #1, , FileID
```

```
   Put #1, , MaxID

   ' Save the position of all Text Controls
   For i = 1 To MaxID
      xPos = TXTextControl1(i).Left
      yPos = TXTextControl1(i).Top
      xSize = TXTextControl1(i).Width
      ySize = TXTextControl1(i).Height
      Put #1, , xPos
      Put #1, , yPos
      Put #1, , xSize
      Put #1, , ySize
   Next i
   Close #1
   ' Save the contents of all TextControls
   For i = 1 To MaxID
      TXTextControl1(i).Save CommonDialog1.filename
   Next i
End Sub
```

The **Load** routine first reads the format identifier and the number of controls. Then it creates the required number of controls, loads their contents and finally moves them to their correct position:

```
Private Sub mnuFile_Load_Click()
   On Error Resume Next

   Dim i As Integer, lFilePos As Long
   Dim FileID As Long, xPos As Single, yPos As Single
   Dim xSize As Single, ySize As Single

   ' Create an Open File dialog box
   CommonDialog1.Filter = "TX Form Demo (*.txf)|*.txf"
   CommonDialog1.DialogTitle = "Open"
   CommonDialog1.Flags = cdlOFNFileMustExist Or _
      cdlOFNHideReadOnly
   CommonDialog1.CancelError = True
   CommonDialog1.ShowOpen

   If Err Then Exit Sub

   ' Open the selected file
```

```
    Open CMDialog1.filename For Binary As #1
    If Err Then
       MsgBox "Can't open file: " + CommonDialog1.filename
       Exit Sub
    End If

    ' Read file header
    Get #1, , FileID
    If FileID <> FILE_ID Then
       MsgBox "Wrong file type: " + CommonDialog1.filename
       Close #1
       Exit Sub
    End If

    ' Destroy existing controls
    For i = 2 To MaxID
       Unload TXTextControl1(i)
    Next i

    ' Create text controls and load their contents
    Get #1, , MaxID
    For i = 1 To MaxID
       Get #1, , xPos
       Get #1, , yPos
       Get #1, , xSize
       Get #1, , ySize
       If i <> 1 Then Load TXTextControl1(i)
       TXTextControl1(i).Move xPos, yPos, xSize, ySize
       TXTextControl1(i).Text = ""
    Next i
    lFilePos = Loc(1)
    Close #1

    For i = 1 To MaxID
       lFilePos = TXTextControl1(i).Load _
          (CommonDialog1.filename, lFilePos)
    Next i
End Sub
```

Printing Multiple Controls

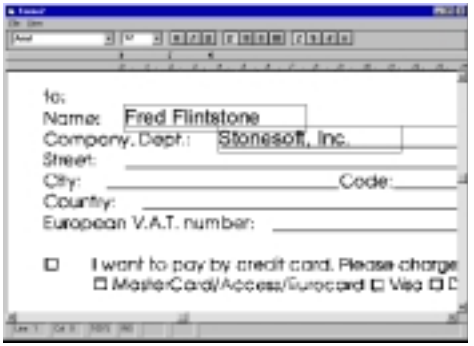
Printing a document is quite straightforward. The **PageWidth** and **PageHeight** properties are set to a value of 0 at design time, so the controls are printed like they are formatted on the screen. The print margin properties are used to specify the positions of the controls on the page.

```
Private Sub mnuFile_Print_Click()
   Dim i As Integer
   Printer.Print
   For i = 1 To MaxID
      TXTextControl1(i).PrintDevice = Printer.hDC
      TXTextControl1(i).PageMarginL = TXTextControl1(i).Left
      TXTextControl1(i).PageMarginT = TXTextControl1(i).Top
      TXTextControl1(i).PrintPage 1
   Next i
   Printer.NewPage
   Printer.EndDoc
End Sub
```

The complete source code of the *Forms1* sample program is contained in the *Forms1* sample source directory.

A Forms Filler

With the *Forms1* sample program, you can place text fields at arbitrary positions on a page. When you print the page, the text fields appear on the paper at exactly the same positions where they were previously placed on the screen. These features will be used in the following sample to create a program for filling out pre-printed forms.



The scanned image of the form is shown in the background of the screen, enabling the user to easily determine the positions of the filled-out fields. He has only to click (with the CTRL key pressed)

on the area of the form where he wants to put text and then start typing. The fields can be moved and resized afterwards by holding down the ALT key and dragging them with the mouse.

The source code for this example is contained in the *Forms2* sample source directory.

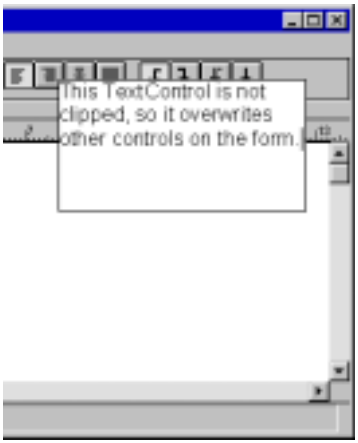
Adding ButtonBar, Ruler and StatusBar

The Button Bar, Ruler and Status Bar are used in a special way in this sample program. If you run the program and click on various fields you will notice that the tools automatically switch to the text field which has been clicked on. This switching is done internally by Text Control, so no programming is required for it. The tools are simply connected to the first member of the Text Control array at design time.

Displaying the Background Image

The background image is displayed by an Image Control. You could also use the Visual Basic PictureBox for this, but the PictureBox can not handle the large image files which result from scanning a full document page, and it does not support the TIFF file format, which is used by most scan programs.

The Image Control is not a separate custom control, but a child window of the Text Control. To display the background image, create a Text Control which has the size of the whole page, and then load an image using Text Control's **ObjectInsertAsChar** method.



The Text Control which displays the background image has an additional function, which again saves a lot of programming work. It acts as a container for the Text Controls which are used as fill-out fields. (A container control enables you to draw other controls within it at design time. Examples of container controls are frames and picture boxes). The big advantage of a container is that it handles all of the clipping for the controls which have been created on

top of it. Otherwise, scrolling the background image would cause the text fields to overwrite anything that lies within the form's boundaries, like ButtonBar, Ruler, and even the scrollbars. It would require many calculations of field positions and sizes and some direct calls to the Windows DLLs on every scroll and resize event to do the clipping without a container control. Using the background Text Control as a container, you need only create the first text field inside of it, and everything else is done automatically.

Working with Transparent Text Controls

Run the program, load a background image and create a few text fields by clicking on this background image. You will notice that the text fields are transparent, so you can see the background image below. Using this feature in a program requires some fine-tuning of the clipping areas with the **ClipChildren** and **ClipSiblings** properties.

These two properties determine which areas of an image are repainted when a new part of a control becomes visible or when its contents have been changed.
For example, if one control is covered by another, it only has to be repainted if the one which lies on top of it is transparent. You will always want to repaint as little as possible to make the application run fast and to avoid unnecessary flickering on the screen. Furthermore you will not want your computer to spend time drawing things which are not visible.

For maximum flexibility in setting the clipping areas and mixing transparent and opaque controls, two properties have been implemented which share this task:

The **ClipChildren** property is used only for Text Controls which act as a container for other Text Controls. When **ClipChildren** is set to True, the areas occupied by the child controls are excluded from the update area. So, if as in the forms filler program, transparent controls are used as children of the container control, this property must be set to False.

The **ClipSiblings** property determines the behaviour between each of the child controls. It must be set to False if the program allows transparent Text Controls to overlap others.

Zooming

Zooming is simply done by setting the **ZoomFactor** property of each of the Text Controls:

```
Private Sub mnuView_ZoomItem_Click(Index As Integer)
   Dim nZoom As Integer, i As Integer
   nZoom = Val(Mid$(mnuView_ZoomItem(Index).Caption, 2))
   TXTextControl2.ZoomFactor = nZoom
   For i = 1 To MaxID
      TXTextControl1(i).ZoomFactor = nZoom
   Next i
   For i = 1 To 5
      mnuView_ZoomItem(i).Checked = (i = Index)
   Next
End Sub
```

Using Marked Text Fields

Marked text fields are markers which are inserted in the text. They can be used to implement a wide range of special functions in a text processor. To name just a few:

- Mail Merge functions

- Spreadsheet-like calculation fields

- Bookmarks

- Automatic table of contents and index generation

- Hypertext viewers which include any kind of buttons, images, pop-up windows or even OLE objects in the text

Any group of characters within the text can be a marked text field. The maximum number of fields is 65,535. Text Control maintains the positions and numbers of the fields. It also takes care of loading, saving and clipboard operations.

A Simple Example

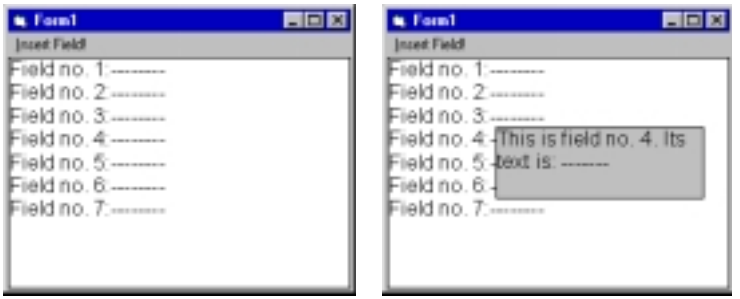
This first sample program will show you how fields are created and what happens when they are clicked on. The code shown here is contained in the *Field1* sample source directory.

The program consists of a form with just one menu item, *Insert Field!*, with an exclamation mark to say that clicking on this item will cause an immediate action instead of dropping a menu. There are two Text Controls on the form, one of which is used as a normal text window (TXTextControl1), the other one as a pop-up window (TXTextControl2).

The following code is executed when the menu item is clicked on:

```
Private Sub mnuInsertField_Click ()
    TXTextControl1.FieldInsert "--------"
End Sub
```

This inserts a field at the current caret position. If you move the cursor over the field, Text Control changes the mouse pointer to an upward pointing arrow (⇧) to indicate that there is something to click on.



If you click on the field, the application receives a **FieldClicked** event, to which it responds by popping up a window which displays the field number.

Only four lines of code are required for this:

```
Private Sub TXTextControl1_FieldClicked(ByVal FieldIndex _
As Integer)
   TXTextControl1.FieldCurrent = FieldIndex
   TXTextControl2.Text = "This is field no." & FieldIndex _
```

```
      & ". Its text is: " & TXTextControl1.FieldText
   TXTextControl2.Move TXTextControl1.FieldPosX, _
      TXTextControl1.FieldPosY
   TXTextControl2.ZOrder
End Sub
```

The first line selects the marked text field which has been clicked on. Line 2 builds the string that is to be displayed in the pop-up window. Line 3 moves the pop-up window, which is initially hidden behind the text window, to the position of the marked text field. Line 4 puts the pop-up window in front of the text window to make it visible. When the mouse button is released, the text window is moved to the front again:

```
Private Sub TXTextControl1_MouseUp(Button As Integer, _
Shift As Integer, X As Single, Y As Single)
   TXTextControl1.ZOrder
End Sub
```

Bookmarks

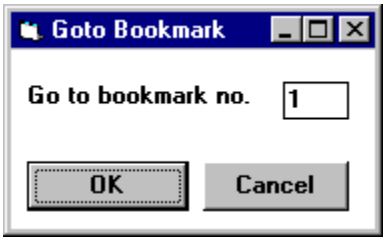
This example shows you how to use Text Control's marked text fields to create bookmarks. The first version will reference the bookmarks simply by their field numbers. The source code for this example is contained in the *Field2* sample source directory.

The sample application has a *Bookmark* menu with two items which are named *Insert* and *Go to...*. Clicking *Insert* creates a marked text field at the current caret position. If a text selection exists, the selected text is converted into a field. If not, the character next to the caret is selected.

```
Private Sub mnuBookmark_Insert_Click()
   If TXTextControl1.Text = "" Then
      MsgBox "Cannot insert bookmark if control is empty."
   Else
      If TXTextControl1.SelLength = 0 Then _
         TXTextControl1.SelLength = 1
      TXTextControl1.FieldInsert ""
   End If
End Sub
```

After typing in some text and inserting a few bookmarks, select the *Go To...* menu item. This will launch a dialog box which allows you to enter

the number of the bookmark to jump to. There is no error processing in this example, so if you enter the number of a non-existent field, nothing will happen.



Clicking the 'OK' button executes the following procedure:

```
Private Sub cmdOk_Click()
   Form1.TXTextControl1.FieldCurrent = Text1.Text
   Form1.TXTextControl1.SelStart = _
      Form1.TXTextControl1.FieldStart - 1
   Form1.TXTextControl1.SelLength = _
      Form1.TXTextControl1.FieldEnd - _
      Form1.TXTextControl1.FieldStart + 1
   Unload Me
End Sub
```

The number which has been entered in the dialog box is taken as a value for the **FieldCurrent** property.

Adding Strings to Marked Text Fields

The source code for this example is contained in the *Field3* sample source directory.

In commercial word processors, bookmarks are normally referenced by names, not just by numbers. The names are typed in by the user when he creates a bookmark. The *Goto Bookmark* dialog box then presents a listbox or combobox in which one of the strings may be selected.

The *Insert Bookmark...* menu item in this version of the program creates a dialog box where the user can enter a label for the bookmark. When the 'OK' button is clicked, the following code is executed:


```
Private Sub btnOK_Click()

   Form1.TXTextControl1.FieldInsert ""
   Form1.TXTextControl1.FieldData( _
      Form1.TXTextControl1.FieldCurrent) = Text1
```

```
   Form1.TXTextControl1.SelLength = 0

   Unload Me

End Sub
```

First, a marked text field is created at the current caret position. Second, the name of the bookmark, which is the text that has been typed in by the user, is stored in the **FieldData** property.

The *Goto Bookmark* dialog box contains a combo box which lists all of the bookmarks which have been created so far. The combo box is filled with the bookmark titles when its form is loaded:


```
Private Sub Form_Load()

   Dim nFieldID As Integer
   nFieldID = 0

   ' Fill the combobox with bookmarks
   Do
      nFieldID = Form1.TXTextControl1.FieldNext(nFieldID, 0)
      If nFieldID > 0 Then
         cboBookmark.AddItem _
            Form1.TXTextControl1.FieldData(nFieldID)
      End If
   Loop While nFieldID <> 0

   ' Copy the first item to the edit control part of
   ' the combo box
   cboBookmark.Text = cboBookmark.List(0)

End Sub
```

When the ‘OK’ button is clicked, the bookmark list is searched for the string which has been selected in the combo box, and the corresponding marked text field is selected.

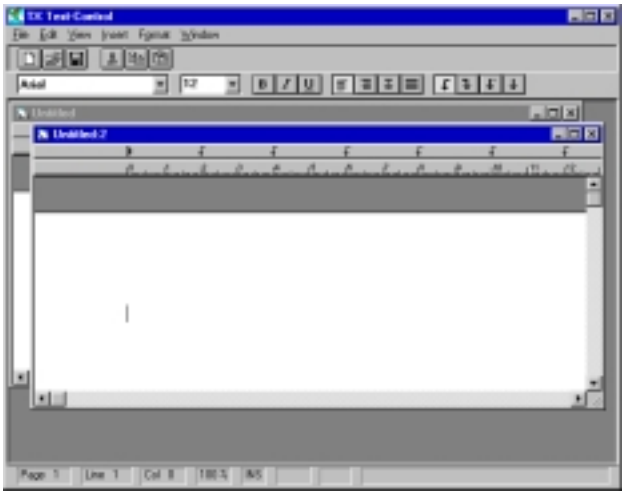
```
Private Sub cmdOk_Click()

   Dim nFieldID As Integer
   nFieldID = 0
```

```
   ' Search for the requested bookmark
   Do
      nFieldID = Form1.TXTextControl1.FieldNext(nFieldID, 0)
      If nFieldID > 0 Then
         If Form1.TXTextControl1.FieldData(nFieldID) = _
            cboBookmark.Text Then
            Exit Do
         End If
      End If
   Loop While nFieldID <> 0

   ' If the bookmark has been found, select it.
   ' Text Control will then automatically scroll to
   ' make it visible.
   If nFieldID <> 0 Then
      Form1.TXTextControl1.FieldCurrent = nFieldID
      Form1.TXTextControl1.SelStart = _
         Form1.TXTextControl1.FieldStart - 1
         Form1.TXTextControl1.SelLength = _
            Form1.TXTextControl1.FieldEnd - _
            Form1.TXTextControl1.FieldStart + 1
   Else
      MsgBox "Bookmark not found."

   End If

   Unload Me

End Sub
```

You can also extend the sample program with a dialog box, similar to the *Go To Bookmark...* dialog, in which a bookmark can be deleted without deleting the text. This would require converting the marked text field to normal text. Use the **FieldDelete** method to achieve this.

More information about marked text fields and a list of all properties, methods and events that can be used with marked text fields, can be found in the Reference part, later on in this manual, in chapter "*Overviews - Marked Text Fields*".

A Word Processor



This chapter shows you how to use Text Control to write a standard word processor. The program is based upon the MDI sample from the Visual Basic Programmer's Guide, with the TextBox controls replaced by Text Controls. If you are not familiar with MDI, control arrays or creating a toolbar you should read that chapter first.

The source code for this example is contained in the *MDIDemo* and *Common* sample source directories.

Adding a PageSetup Dialog Box

The *Page Setup* dialog box is used to determine the page size and print margins. The maximum page size is restricted by the capabilities of the default printer. For implementation details, look at the source code of the PageDlg form.

A Print Dialog Box

When the *Print...* menu item is clicked, first a Common Dialog box is shown to let the user enter the range of pages, number of copies and printer specific information. The rest of the procedure, which is part of the MDIChild form, is just a loop

which sets the appropriate Text Control properties for every page to be printed.

Search and Replace

Searching and replacing is entirely done in Text Control. You just have to assign a value of 1 for *Search* or 2 for *Search And Replace* to the **FindReplace** method. Text Control then opens the Windows Common Dialog box.

Using Paragraph Frames

With Text Control, you can add lines and frames to a paragraph or a range of paragraphs. For instance, you can put a line at the bottom of a caption like in the header of this manual.

The dialog box for paragraph frames is not included in the Text Control, but the source code is included in the MDI sample.

The properties which are responsible for paragraph frames are **FrameDistance**, **FrameLineWidth**, and **FrameStyle**.

Dialog Boxes for Text and Background Color

This is also done with Common Dialogs. The color value returned from the dialog box is assigned to the **ForeColor** or **BackColor** properties.

Using Text Control as a Bound Control

This chapter describes how to use Text Control to access databases with the Visual Basic Data Control. If you are not familiar with the Data Control or with Bound Controls in general please refer to the Visual Basic documentation.

The source code for this example is contained in the *Data* sample source directory.

Connecting a Text Control to a Data Control enables you to store the contents of the Text Control as a record in a database. Not only is the plain text stored, but also all formatting information, e.g. font and paragraph attributes, colors and image file names. The data is stored in a binary format which is the same as that used by the **Load** and **Save** methods.

The *Data* sample program is connected to a small database which contains descriptions of some of Text Control's properties. The database was created with the Visual Basic Data Manager and then filled by inserting text from the clipboard. You can browse through the records of the database by clicking the Data Control buttons on the lower left side of the window. If you change something in the current record, the changes will automatically be written to the database as soon as you click on one of the buttons.

Storing a Text Control's contents with all formatting information as is illustrated in this example requires the **DataTextFormat** property to be set to *1 - Binary*. In the default mode, which is *0 - Text*, only the text is stored. The *0 - Text* mode can be used to access databases which have been created by other programs which do not use the Text Control data format.

Calling DLL Functions from Visual Basic Code

Sometimes it is necessary to access the Text Control DLL directly instead of using the OCX Properties. There are messages which, because most Visual Basic users will never need them, have no corresponding properties, but which may be useful for your program.

The *CallDLLs* sample program, whose source code is contained in the *CallDLLs* sample source directory, shows you how to use these messages. You may want to browse through the DLL Reference online

help file to see which other messages might be useful. The numbers of the Text Control messages are listed in \samples\dll\inc\tx.h.

Inserting Objects

This sample program shows you how to insert external objects into a Text Control. The source code for this example is contained in the *Objects* sample source directory.

An external object can be anything that has a window handle, for instance buttons, list boxes, combo boxes, or other Text Controls. This feature enables you to create active documents, in which the user can enter data, select items from lists, or press buttons. Imagine a pizza order form, where you enter your name and address into the Text Control fields and select the things you want your pizza to consist of from various drop-down menus.

How it Works

In this sample, two kinds of external objects can be inserted: Text Controls, which act as text input fields, and combo boxes, which can be used to select an item from a list. The two kinds of objects are control arrays. One element of each of the controls is placed on the Text Control at design time is made invisible and is assigned an **Index** property value of 0. At run time, when the user clicks a menu item to create an object, a new element of the respective property array is created, made visible, and inserted into the Text Control. Objetcts are inserted using the **ObjectInsertAsChar** or **ObjectInsertFixed** Method. Like images, objects can be inserted 'as character', making them act as if they were characters within the text and move as the text changes, or 'fixed', in which case the text flows around the objects. The last parameter of the **ObjectInsertAsChar** or **ObjectInsertFixed** methods is used to disinguish between different kinds of objects.

Loading and Saving

When you save a document which contains external objects, then the Text Control does not know what data these controls may contain. To enable you to save the object's data, Text Control sends an

ObjectGetData event for every object to be saved. When you then re-load the document, the saved data is passed back as a parameter of **ObjectSetData** events. Note that you do not need to process these events for inserted Text Controls.

Using the Clipboard

Another important event is **ObjectGethWnd**. This event is sent whenever an object is to be created, i.e. when a document which contains objects is pasted from the clipboard, or loaded from a file. In response to this event, the application creates the object and returns its window handle.

Note that you cannot use the **SizeMode** property to move an inserted Text Control with the mouse. Using **SizeMode**, only the Text Control's window is moved, but not the OLE frame. If your application requires Text Controls to be moved, insert them by calling the **CreateTextControl** function, which is described in the DLL Reference Manual.

For an example of how to use the **Objectxxx** properties amd methods, refer to the MDI sample program.

Mail Merge

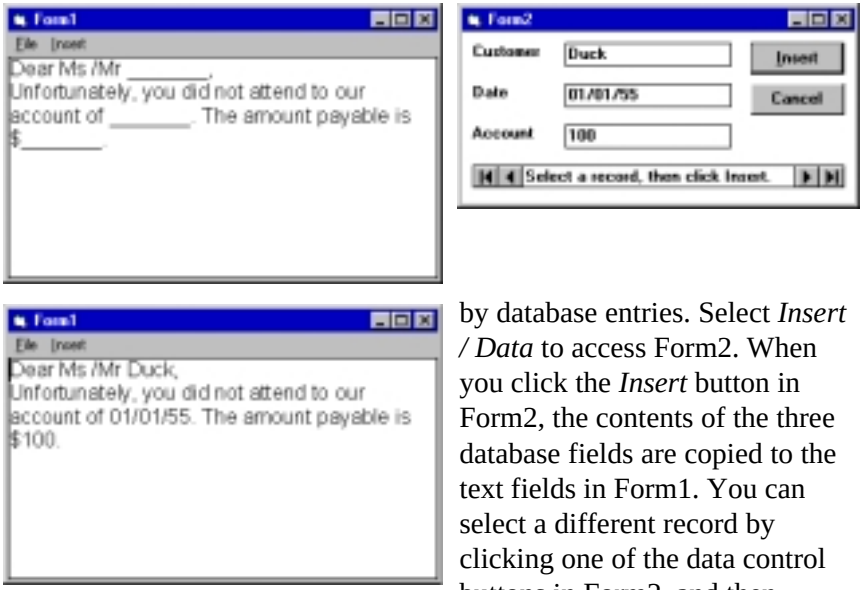
The chapter "*Using Text Control as a Bound Control*" showed you how to store a Text Control's entire contents in a database field. For implementing functions like mail merge, however, the requirements are different: the contents of database fields have to be inserted at specified positions in a previously prepared document. The following sample program provides you with the basis of how to this.

The code shown here is contained in the *Stdlet* sample source directory.

The Sample Program

The program consists of two forms, Form1 for creating a text and Form2 for connecting it to the database.

Start the program and use the *File / Open...* command to load the sample file 'account.tx'. The file contains three fields which are to be replaced



by database entries. Select *Insert / Data* to access Form2. When you click the *Insert* button in Form2, the contents of the three database fields are copied to the text fields in Form1. You can select a different record by clicking one of the data control buttons in Form2, and then clicking *Insert* again to replace the fields.

How it Works

Each of the three edit controls in Form2 are connected to a field in the database. The edit controls are used as bound controls, so when you browse through the database by clicking on the data control buttons, the contents of the selected database record are automatically copied to the edit controls. The only thing left to do is to copy the data from the edit controls to the text fields in the document. This is done when you click on the *Insert* button:

```
Private Sub cmdInsert_Click()
   Form1.TXTextControl1.FieldCurrent = 1
   Form1.TXTextControl1.FieldText = Form2.Text1
   Form1.TXTextControl1.FieldCurrent = 2
   Form1.TXTextControl1.FieldText = Form2.Text2
   Form1.TXTextControl1.FieldCurrent = 3
   Form1.TXTextControl1.FieldText = Form2.Text3

   ' Uncomment this to send the result to the printer.
   ' Printer.Print
```

```
    ' Form1.TXTextControl1.PrintDevice = Printer.hDC
    ' Form1.TXTextControl1.PrintPage 1
    ' Printer.EndDoc
End Sub
```

To implement a real mail merge function you will have to add a dialog box in which the user can select the database to be used. You may also want to provide a variable number of database fields which are dependent on the contents of the selected database.

Using Hypertext Links

This chapter shows how to use Text Control's marked text fields to insert hypertext links and targets into text documents and how to respond to events which Text Control fires when the user clicks on a hypertext link.

The source code for the following examples is contained in the subfolders *Step1* to *Step4* of the *HyperLnk* sample source directory.

Step 1: Inserting a Hypertext Link

In this first sample program a hypertext link will be inserted in a text document. The document is saved then as a HTML file so that it can be viewed in a browser.

Hypertext links are handled as a special type of a marked text field. A hypertext link therefore is inserted by calling the **FieldInsert** method, and then specifying the type of the field with the **FieldType** property:

```
TXTextControl1.FieldInsert "Text Control Web Site"
TXTextControl1.FieldType(TXTextControl1.FieldCurrent) = _
        txFieldExternalLink
```

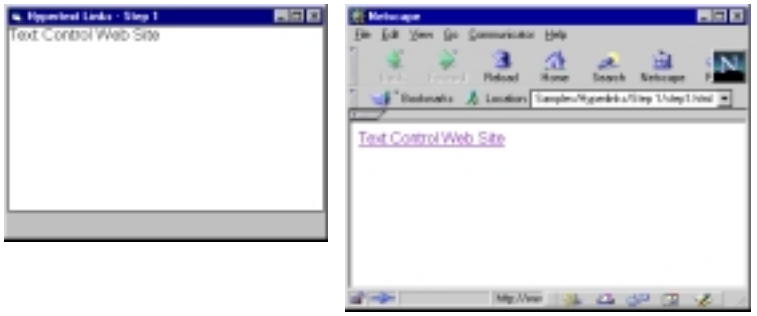
To store the target to where the link points, the **FieldTypeData** property is used:

```
TXTextControl1.FieldTypeData(TXTextControl1.FieldCurrent)= _
                "http://www.textcontrol.com"
```

The following line of code saves the document, containing the hypertext link, which has just been inserted as a HTML file in the sample folder:

```
TXTextControl1.Save App.Path & "\step1.html", , 4, 0
```

When this file is loaded with a web browser, the hypertext link will be displayed as specified in your browser's settings. Clicking on the link, will take you to the Text Control web site.



Note that there is no code for the **Click** events yet, so clicking on the hypertext link in the Text Control will have no effect. Also, the link is neither underlined nor colored.

Step 2: Adding a Dialog Box for Inserting Hypertext Links

In this second sample program a dialog box is created which enables the user, to insert hypertext links in a more convenient way. Additionally, hypertext links which have previously been inserted or loaded from a file, can be edited and modified. Note that, while hypertext links are usually associated with HTML files, they can as well be stored in RTF or Microsoft Word files, or in Text Control's proprietary format.

The dialog box has two text boxes. The first is for the text that represents the hypertext link in the document and the second is for the address, to where the link points. In the step 1 example, the representing text was "*Text Control Web Site*", and the address, to where the link points, was "*http://www.textcontrol.com*".

The same dialog box is used for both, inserting a new and editing an existing hypertext link. Depending on whether the current input position is inside of an existing link, this link is modified. Otherwise a new one is inserted.

The dialog form's properties, *tx* and *bShowHyperlinks*, are used to pass a Text Control's reference and some information about how to display the hypertext links to the form.

```
Public Sub do_mnuInsert_Hyperlink_Click(tx As TXTextControl,
bShowHyperlinks As Boolean)
   Set frmHyperlink.tx = tx
   frmHyperlink.bShowHyperlinks = _
      (mnuView_Hyperlinks.Checked = True)
   frmHyperlink.Show 1
End Sub
```

When the form is loaded, the text boxes are filled with the text and link information when the current input position is inside of an existing link:

```
Private Sub Form_Load()
   If CaretInsideHyperlink(tx) <> 0 Then
      txtLinkedText = tx.FieldText
      txtLinkTarget = tx.FieldTypeData(tx.FieldCurrent)
   Else
      txtLinkedText = tx.SelText
   End If
End Sub
```

The user then can change the displayed information. The information is then transfered to the document by either inserting a link or modifying the existing one when the 'OK' button is pressed:

```
If tx.FieldAtInputPos <> 0 Then
   ' editing an existing hypertext link
   tx.FieldText = txtLinkedText
   tx.FieldType(tx.FieldCurrent) = txFieldExternalLink
   tx.FieldTypeData(tx.FieldCurrent) = txtLinkTarget
Else
   ' insert new hypertext link
   tx.FieldInsert txtLinkedText
   tx.FieldType(tx.FieldCurrent) = txFieldExternalLink
```

```
   tx.FieldTypeData(tx.FieldCurrent) = txtLinkTarget
   HighlightHyperlinks tx, bShowHyperlinks
End If
```

Finally, there is a menu item to switch the character format of the hyperlink's text to blue colored and underlined style. The menu item calls the function **HighlightHyperlinks**, which is defined in the file HyperlinkFunctions.bas.

Step 3: Adding Targets

Step 1 and 2 only handle references to external resources, i.e. addresses of web pages or files. In this step, links to positions in the same document will be handled. These links are called internal links and the positions, to where they point, are called targets. Targets are also refered to as anchors (in the context of HTML editors) or bookmarks (in word processors). When using this example, first add some text and then some targets with the *Insert / Target...* menu item. Finally use the *Insert / Hypertext Link...* menu item to add links to these targets.

Inserting a Target

Targets are realized again as a special type of a marked text field. The type and the target's name must be set with the **FieldType** and the **FieldTypeData** properties. Unlike links, targets have no visible text, therefore an empty field must be inserted with the **FieldInsert** method to insert a target:

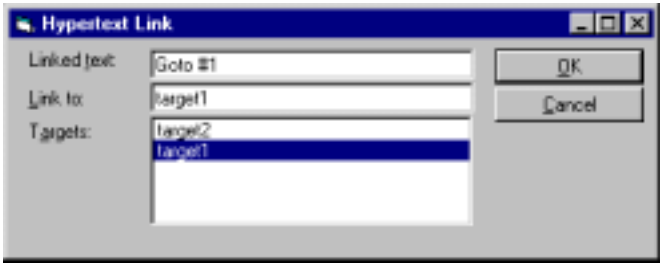
```
Dim TargetName As String
TargetName = InputBox("Target name:", "Insert target")
If TargetName <> "" Then
   TXTextControl1.FieldInsert ""
   TXTextControl1.FieldType(TXTextControl1.FieldCurrent) = _
        txFieldLinkTarget

TXTextControl1.FieldTypeData(TXTextControl1.FieldCurrent)_
       = TargetName
End If
```

Only one text box is required to display the name of a target, so a simple InputBox statement can be used.

Inserting Links to Targets

To insert links to the just inserted targets, the *Hypertext Link* dialog box is extended with a list box showing the names of all targets the document contains. The **FieldNext** method is used to fill this list box:

```
Dim FieldID As Integer
List1.Clear
FieldID = tx.FieldNext(0, &H100&)
While FieldID <> 0
List1.AddItem tx.FieldTypeData(FieldID)
   FieldID = tx.FieldNext(FieldID, &H100&)
Wend
```



When the user selects a target, the *Link To* field is filled with the target's name. After typing the link's text and pressing the 'OK' button, the link is inserted. An internal link is inserted in the same way as the external links from step 1, but the **FieldType** property now is set to **txInternalLink** and the **FieldTypeData** property is set to the target's name.

Jumping to a Target

After inserting internal links and targets, a jump must be realized. When the user clicks on a marked text field that represents a hypertext link, Text Control fires a **FieldLinkClicked** event. The information provided through this event can be used with the **FieldGoto** method to jump to the target:

```
Private Sub TXTextControl1_FieldLinkClicked(_
            ByVal FieldId As Integer, _
            ByVal FieldType As Integer, _
```

```
                TypeData As String)
   TXTextControl1.FieldGoto txFieldLinkTarget, TypeData
End Sub
```

While the **FieldGoto** method is used for targets within the same file, links to external targets must be treated differently. When the **FieldLinkClicked** event occurs, and the **FieldType** parameter indicates that the link is external, then it depends on the type of the application, what to do. External links can point to, for instance, files on the local harddisk, or addresses in the internet.

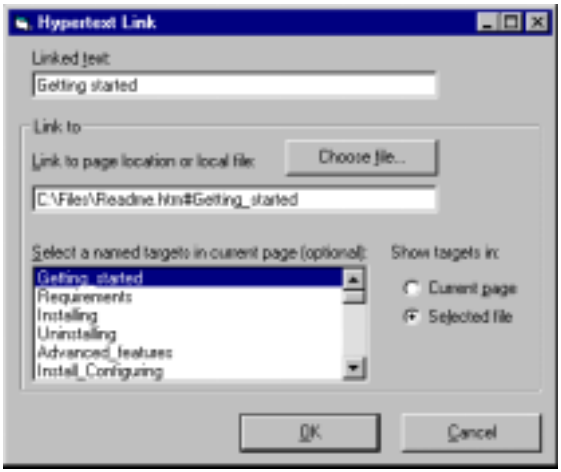
Note that responding to the events is only required for making the hypertext links work while the text is edited in Text Control. If the text is saved to a file and displayed with a browser, then the hypertext links will work depending on the used browser.

Step 4: Adding Jumps to External Targets

Finally, in this step, jumps to other documents and jumps to targets in these documents are added.

An Enhanced Dialog Box for Displaying and Selecting Targets

Again the *Hypertext Link* dialog box is extended to choose an external file. A *Choose File...* button is placed on the form that triggers a common dialog. After the user has chosen a file, its name is displayed in the text box and the file is searched for internal targets:




```
Private Sub CheckFileForTargets(file As String)
   tx(1).LoadSaveAttribute(txEnableLinks) = True
   tx(1).Load file, 0, 4
   FillListboxWithTargets (1)
```

```
   optSelFile.Value = True
   loadedFile = file
End Sub
```

For this purpose the file is loaded in a second, invisible Text Control. Then the **FieldNext** method is used as in step 3 to list all targets.

Jumping to an External Target

To implement the jump to an external link, the code added to the **FieldLinkClicked** event in step 3 must be extended. The following code does not handle jumps to internet addresses, it only implements jumps to targets in other files. To seperate a file from a name of a target, Text Control uses the '#' character. The following code separates the file name and the target's name, loads the file with the **Load** method and jumps to the target with the **FieldGoto** method:

```
ElseIf FieldType = txFieldExternalLink Then
   'This sample does not feature links to www sites,
   'so exit sub
   If Left(TypeData, 7) = „http://“ Then
      Exit Sub
   End If

   TXTextControl1.LoadSaveAttribute(txEnableLinks) = True
   pos = InStr(TypeData, „#“)

   'File name includes an internal link
   If pos >= 0 Then
      'Ask the user to save changes, if any
      If bDocDirty Then
         ret = MsgBox(„Save changes?“, vbYesNoCancel, _
         „Question“)
         If ret = vbYes Then
            mnuFile_SaveAs_Click
         ElseIf ret = vbCancel Then
            Exit Sub
         End If
      End If

      'Extract file name & path from full path
```

```
            str = Left(TypeData, pos - 1)
            TXTextControl1.Load str, 0, 4
            'Extract file position from full path
            str = Mid(TypeData, pos + 1)
            TXTextControl1.FieldGoto txFieldLinkTarget, str
         Else
            'File name doesn't include an internal link
            TXTextControl1.Load TypeData, 0, 4
         End If
      End If
```

Loading and Saving Files containing Hypertext Links

When an HTML, RTF or Microsoft Word document is loaded, Text Control must convert containing hypertext links to appropriate marked text field, as described above. To perform this, a programmer must set the **LoadSaveAttribute**(**txEnableLinks**) before using the **Load** method. Otherwise hypertext links and target fields are not converted. When a document is saved, marked text fields that represent hypertext links, are always converted to the appropriate format.

If Text Control's proprietary format is used, setting **LoadSaveAttribute** is not necessary.

More information about hypertext links and a list of all properties, methods and events that can be used with marked text fields, can be found in the Reference part, later on in this manual, in the chapter "*Overviews - Marked Text Fields - Special Types of Marked Text Fields*".

Headers and Footers

This example shows how to use headers and footers. The source code is contained in the *Headers* sample source directory.

TX supports headers as well as footers. You also have the ability to create a different header or footer for the first page.

To insert a header or footer in the example, click on *Insert* and choose one of the four possible options. The code that is executed when

clicking on one of the menu items is almost the same. For the *Header* menu item it looks as shown below. The line

```
TXTextControl1.HeaderFooter = TXTextControl1.HeaderFooter +
txHeader
```

informs Text Control that a header should be added to the current settings.

Setting the **HeaderFooterStyle** property to **txMouseClick** enables the user to activate the header with a single click rather than a double-click. Activating a header or footer with a double-click is Text Control's default setting. More information about how to use headers and footers and a list of all properties, methods and events that can be used with headers and footers, can be found in the Reference part, later on in this manual, in the chapter "*Overviews - Headers and Footers*".

When using properties, Text Control distinguishes between the main text and headers or footers. To switch between these different independent text parts, Text Control provides the **HeaderFooterSelect** method:

```
TXTextControl1.HeaderFooterSelect  txHeader
TXTextControl1.SelText = "Header"
TXTextControl1.HeaderFooterSelect  0
```

This code selects the header, so that the following code affects the header and then sets the headers text. Finally the mode is reset to zero using the **HeaderFooterSelect** method. More information about programming with headers and footers see the chapter "*Overviews - Headers and Footers - Programming Headers and Footers*".

A header or footer is activated from programming code using the **HeaderFooterActivate** method. To delete a header or footer, simply substract the **txHeader** constant from the current **HeaderFooter** settings.

The following is the complete code of the menu item:

```
Private Sub mnuHeader_Click()
   If mnuHeader.Checked = False Then
      TXTextControl1.HeaderFooter = _
         TXTextControl1.HeaderFooter + txHeader
```

```
      TXTextControl1.HeaderFooterSelect txHeader
      TXTextControl1.SelText = "Header"
      TXTextControl1.HeaderFooterSelect 0
      TXTextControl1.HeaderFooterActivate txHeader
      mnuHeader.Checked = True
   Else
      TXTextControl1.HeaderFooter = _
         TXTextControl1.HeaderFooter - txHeader
      mnuHeader.Checked = False
   End If
End Sub
```

Drag and Drop

This example shows how to use the **InputPosFromPoint** method to realise a simple Drag&Drop in a Text Control application.

Drag&Drop in a text editor enables the user to drag a piece of text and drop it in a new location of the document. So, the incoming mouse events have to be analyzed and handled.

In the **MouseDown** event, the **InputPosFromPoint** method is used to get the character position the user has clicked on. The current input position and the length of the selection are stored in global variables, because they are needed in the **MouseUp** event. If the input position the user has clicked on, is inside of the current selection, dragging can be started. First a global variable named *dragging* is set to true and the **MousePointer** property is changed to indicate that dragging is in process. The text and format information of the current selection is copied to a memory buffer using the **SaveToMemory** method. Finally, the Text Control's **EditMode** property is set to *2 - read only*.

```
‘Get current input position and the current selection
pos = TXTextControl1.InputPosFromPoint(X, Y)
gblStart = TXTextControl1.SelStart
gblLength = TXTextControl1.SelLength

‘Check if the click occured in the current selection
If gblStart <= pos And gblStart + gblLength > pos Then
```

```
‘Start dragging
   data = TXTextControl1.SaveToMemory(3, True)
   dragging = True
   MousePointer = 2
   TXTextControl1.EditMode = 2
End If
```

In the **MouseUp** event procedure the **InputPosFromPoint** method is used again to get the character position where the user has left the mouse button. When dragging is in process and the input position is not inside the current selection, the drop opertation can be performed. The previously saved text now is inserted with the **LoadFromMemory** method after setting the new input position with the **SelStart** property.

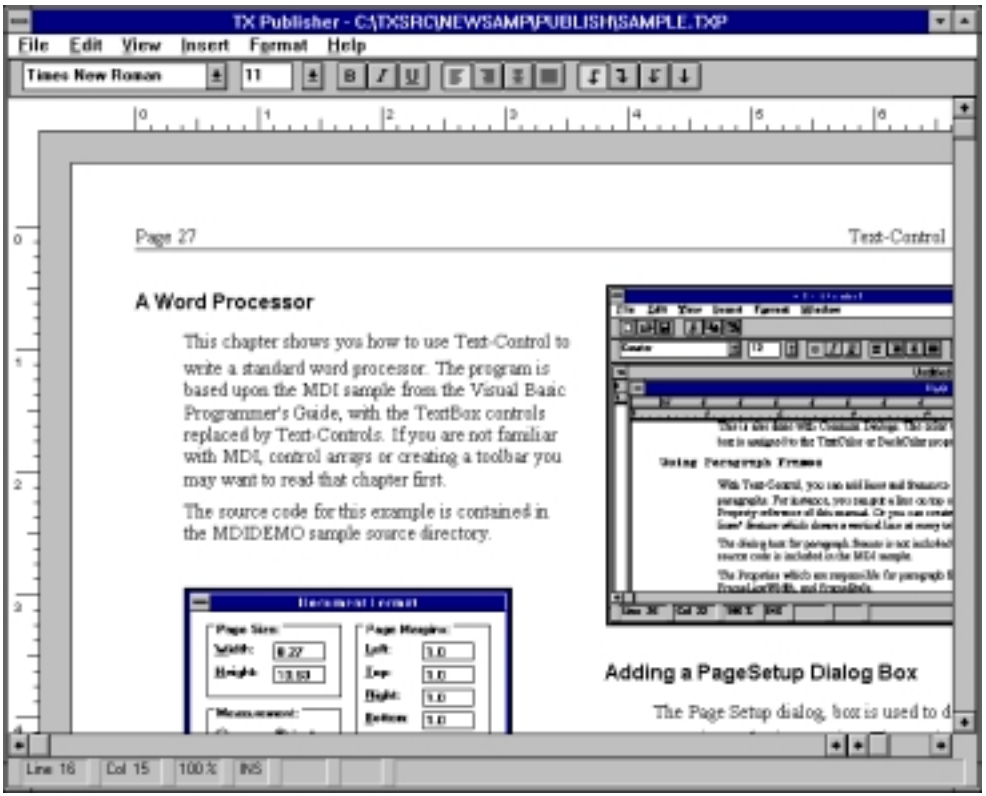
```
pos = TXTextControl1.InputPosFromPoint(X, Y)
If dragging Then
   ‘Check if the new input position is outside of
   ‘the current selection. If it’s not, do not
   ‘copy the text
   If Not (gblStart<=pos And gblStart + gblLength>pos) Then
      TXTextControl1.SelText = „“
      If pos < gblStart Then
         TXTextControl1.SelStart = pos
      Else
         TXTextControl1.SelStart = pos - gblLength
      End If
      TXTextControl1.LoadFromMemory data, 3, True
   End If
   ‘End dragging
   dragging = False
   MousePointer = vbNormal
   TXTextControl1.EditMode = 0
End if
```

TX Publisher - An Advanced Example

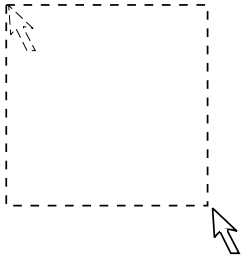
This sample progam is written entirely in Visual Basic, with no third party custom controls or DLLs except those included with Visual Basic itself. The program is intended to be used as a starting point for your applications, and it contains all the basic functions like loading and saving documents, printing, zooming, as well as the scroll interface. You can easily add more features and customize the program without having to start from scratch.

Text Frames and OLE Objects

TX Publisher works with text frames. This can entail pure text frames into which new text is entered or OLE objects. The type of frame is



defined in the Insert menu via the 'New Frame' menu item. In principle, the handling of text and OLE frames is the same. We will explain the frame handling using examples with text frames, and will then deal with the OLE object.



Drawing Text Frames

In order to draw a frame, click on an empty part of the page and, depressing the left mouse button, drag the mouse down and to the right. If you require a different page display for this, select it in the View menu via the 'Zoom' menu item.

The borders of the newly created text frame can be made visible by selecting 'Text Frames' in the 'View' menu. A paragraph ruler can be shown above the Text Frame. This setting is likewise made in the view menu, using the menu item ‘Paragraph Ruler’.

Text can now be entered into the newly created frame until it is full, at which point the frame has to be enlarged or the next frame has to be created. Alternatively you can draw all the required text frames successively, and then start entering text. Note that you can only start entering text in the first frame.

Connecting Text Frames

The text frames are linked automatically. This means that text automatically flows from the current frame into the next frame when the current frame is full.

If you click on an empty text frame which is further down the chain, then the cursor will stay in the last window which contains text.

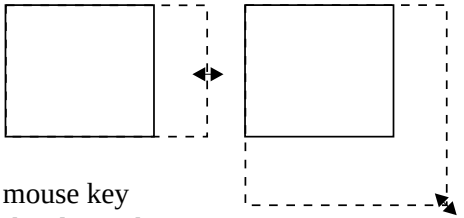
The text frames are numbered internally in their sequence of creation.

Deleting and Creating Frame Connections

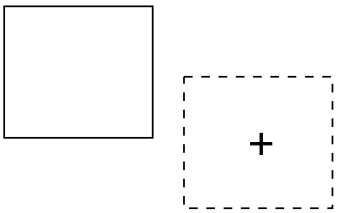
You can eliminate the connection between frames and, if required, regenerate them. You delete the connection to the following frame by clicking on the respective frame with the CTRL key depressed, and by answering the subsequent question displayed, ‘Delete connection to next window’, with Yes.

In order to create a connection, click on the frame to be connected to and keep the mouse button depressed until a symbol with a sheet of paper in a hand is displayed. Keeping the CTRL key and mouse button depressed, drag the symbol onto the frame to which you wish to create a connection. Answer the following question displayed, ‘Connect Frame No. x to Frame No. y’, with Yes. If it is not possible to create a connection, an error message will appear.

Changing Frame Size and Position

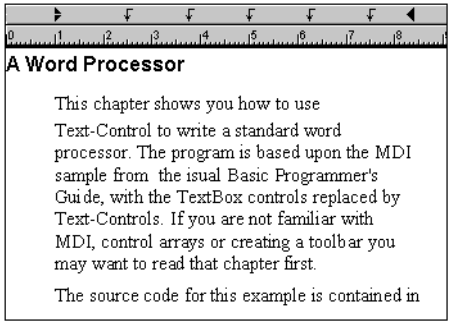


The size and position of a text frame can be changed subsequently. To change the size, click the frame borders or a corner of the frame with the ALT key depressed. Keep the mouse key depressed, and drag the respective border to the desired position.



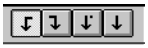
To change the position, click at any position within the frame with the ALT key depressed. Whilst keeping the mouse button depressed, drag the frame to the desired new position.

Setting Indents and Tabs



The currently active frame receives a paragraph ruler when this feature has been activated in the 'View' menu. Using the paragraph ruler you can set indents and tabs.

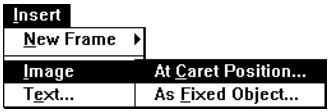
Indents can be changed by sliding the two small triangles on the left side of the ruler, and the large triangle on the right.



Tabs are left-aligned by default. To create right-aligned, decimal or centered tabs the tab type can be selected using the Button Bar.

You can set tabs by clicking at the desired position on the paragraph ruler. You can then shift the tab marker by clicking on it, and simultaneously dragging it along the ruler with the mouse button depressed. You can remove a previously set tab by pulling it downwards, away from the ruler. The maximum number of tabs is 14.

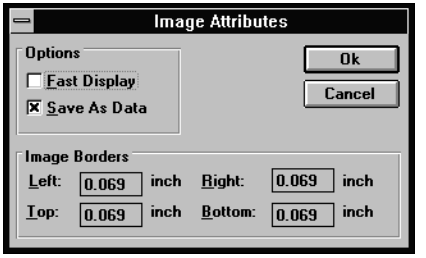
Using Images



Images can be inserted via the 'Insert/ Image' menu item or from the clipboard. The menu lets you choose between inserting the image 'At Caret Position' or 'As Fixed Object'. Images which are inserted at the caret position are treated like characters, and they move with the text as it changes.

When inserted as fixed objects, images have a fixed position on the page, and the text flows around them. The initial position of an image inserted in this way is one inch from the top left corner of the page. You can move it to the desired position just like you move text frames, which is by depressing the ALT key and dragging the image with the mouse.You can also change the size of the image in this way.

Clicking on an image and selecting 'Image...' from the 'Format' menu lets you select image attributes in a dialog box. You can adjust the size of the borders, i.e of a frame around the image where no text is displayed, and you can select if you want the image data to be included in your document file or if you just want to store a file reference. Storing the image data increases the size of your document file, but has the advantage of making the document independent of additional image files.

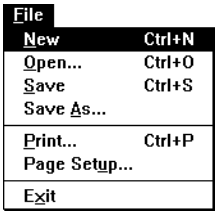


Images which are inserted from the clipboard are always inserted 'at caret position' and saved 'as data'.

OLE Objects

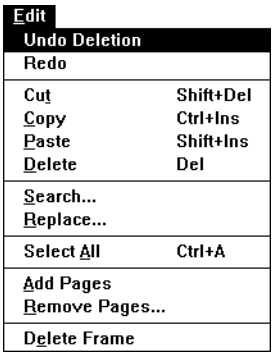
If you select 'OLE Object' in the 'Insert / New Frame' menu, frames are created in the following way. The frame is drawn as described above, by placing the mouse at the top-left corner of the frame to-be and dragging it down and towards the right. A dialog box entitled ‘Insert Object’ then appears. This dialog box also appears on the screen if you click over the frame with the right mouse button. You have the choice of creating a new object or of loading a file.

The File Menu



In the File menu you will find standard functions such as: New, Open, Save, Save As, Print, Page Setup, Exit. These will be familiar to you from various other Windows applications and will therefore not be described in more detail at this point.

The Edit Menu

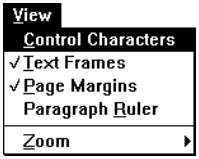


The Edit menu also includes a number of standard functions including Undo and Redo function, Cut, Copy, Paste, Delete, Search, Replace, and Select All. Regarding the Undo function, three different actions can be undone; Input, Deletion and Formatting.

Other menu items include ‘Add Pages’ and ‘Remove Pages’, with which you can insert and delete pages. When creating a document there are two document pages. In order to create additional pages, select Add Pages. Two further pages are then added to the existing ones. Using the small scroll bar at the bottom right, you can flick through the pages. You can delete the last two pages using 'Remove Pages'.

If you wish to delete a frame, initially activate it by clicking on it, and then select ‘Delete Frame’. After agreeing to ‘Delete Text Frame x’, the respective frame is removed.

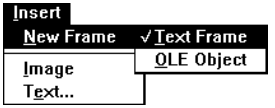
The View Menu



In the View menu you switch in or switch out one or more of the displays of Control Characters, Text Frames, Page Margins and Paragraph Rulers. You can also set the display size of the page view. Using ‘Zoom’ you have the following options available: Full Page, 30%, 50%, 75%, 100%, 200%.

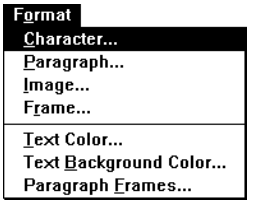
Regarding the Control Characters, soft and hard line breaks and blanks are displayed.

The Insert Menu



In the Insert menu you determine, via 'New Frame', the type of frame to be created. You can choose between 'Text Frame' and 'OLE Object'. For this purpose, read the previous pages. Using the ‘Image’ menu item, a picture can be imported, and by selecting ‘Text’, ASCII or RTF text files can be inserted at the current caret position.

The Format Menu



In the Format menu you can perform character and paragraph formatting. You can determine the text colour and text background colour, and using the menu item ‘Paragraph Frames’, you can define lines or frames for paragraphs.

The Help Menu

Using ‘Help Topics’ you call up the Online help service. You can also view an info window via the menu item ‘About TX Publisher’

How the Program Works

Much of the program's functionality is based on the concept of container controls. At the bottom of the control hierarchy there is a page ruler, which is placed directly on the form. On top of the page ruler there is a picture box which acts as a container for the document pages, which are themselves picture boxes. Finally, the text frames, OLE frames and the paragraph ruler use the page controls as containers. Although this may seem a bit complicated at first sight, it saves you a lot of programming work, because this approach helps to divide the program into logical blocks, and handles all the different clipping regions.You can see how the controls are put together when you look at the program in Visual Basic design mode. (See next page).

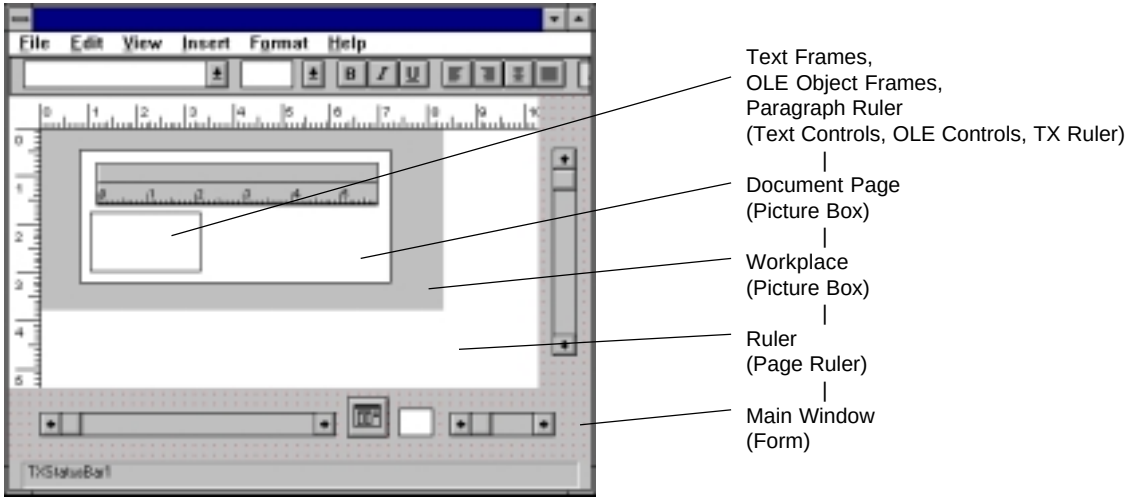
When the program is started, two document pages are created in a default size of A4 or Letter, depending on the system's country setting. The size of the workspace is then automatically adjusted so that the two pages can be shown side by side with a gray border around them. Settings which do not change during the program execution are made in the main form's Load event, whereas settings which depend on the window size or the zoom factor are made in the form's Resize event.

Managing Global Data

Most of the global data is managed by the controls themselves and thus does not have to be stored explicitly in variables.
For instance, the position and size of the text frames are stored in the control's Left, Top, Width and Height properties. Information which cannot be stored in control properties has been collected into a single global structure. This structure is called 'Doc' and contains information about zoom factor, page margins and the total number of text and OLE frames in the document. Its definition is to be found in the file 'global.bas'.

Creating new Text and OLE Frames

A new frame is created when the user draws a rectangle on the page. This happens in three stages in response to the page control's mouse events:



On picPage_MouseDown, the mouse coordinates are stored as the top left corner of the new control.

On picPage_MouseMove, a rectangle is drawn showing where the new control will be placed after the mouse button has been released.

Finally, when the picPage_MouseUp event occurs, the rectangle is deleted and a Text Control or OLE control is created at its coordinates.

The Text Controls and OLE controls, as well as the document pages, are implemented as control arrays, so a new instance of one of them can be created by calling the Load function. The newly created control is by default a child window of picPage(0), which is the element of the page control array that was created at design time. To have the control displayed on top of the current page, the Windows API function SetParent is used.

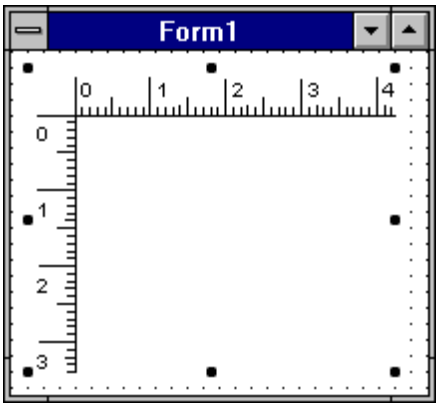
Connecting Text Frames

The last step in creating a new text frame is to connect it to its predecessor so as to enable text to flow from one control to the next. This is simply done by assigning the window handle of the new Text Control to its predecessor's NextWindow property. The connection can be deleted later on by setting the property to a value of 0.

Deleting frames

A frame is deleted when the user selects the 'Delete Frame' menu item. This does not really remove it from the control array, but simply makes it invisible and marks it as deleted by setting its Tag property to a value of -1. The frames marked in this way are removed when the document is saved to disk. The Tag property normally contains the number of the page to which the control belongs.

The Page Ruler Control



When you first start TX Publisher you will notice that the ruler looks different from the one in the standard version of TX Text Control. The ruler is in fact an additional custom control. Its filename is ‘PgRul.Ocx', which is short for ‘Page Ruler’.

The Page Ruler control can be used as a container for other controls. In the TX Publisher sample program, it is used as a document page, on which the text frames are placed. A detailed description of the Page Ruler's properties, methods and events can be found in the Reference part of this manual.

Delphi User's Guide

Creating a Simple Word Processor

This chapter shows you how to create a small word processor from scratch with just a few lines of code. It will be able to load and save files, use the clipboard, and will have dialog boxes for character and paragraph formatting, a ruler, a status bar and full keyboard and mouse interface.

The source code for this example is contained in the *Simple* sample source directory.

Creating the Project

Assuming that you have already run the Text Control installation program and started Delphi, the next step is to create a project for the text processor. To do this begin by selecting the *New Application* command from the file menu. If you have already imported Text Control into Delphi, its icons are shown when the ActiveX tab is selected. Otherwise, click on *Controls / Import ActiveX...* and choose TX Text Control from the given list. Click *Install* and then *OK* until all dialog boxes have been closed. Now you will see the following four additional icons when the ActiveX tab is selected:

 The Text Control Icon

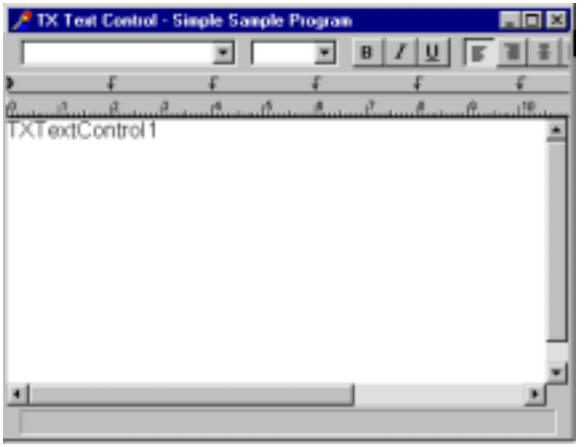
 The Status Bar Icon

 The Button Bar Icon

 The Ruler Icon

Creating the Controls

The next step is to put these four controls on a form and connect them. Run Delphi and create a new project. Select the ´OCX´ page in the component palette to have the 4 Text Control icons displayed. Click on the Text Control icon and draw it on the form. In the same way, create a Ruler and a Button Bar on top of the Text Control, and a Status Bar below it. Your form should now look like the diagram on the right:



Connecting the Controls

Add the following code to the form's FormShow Event procedure:

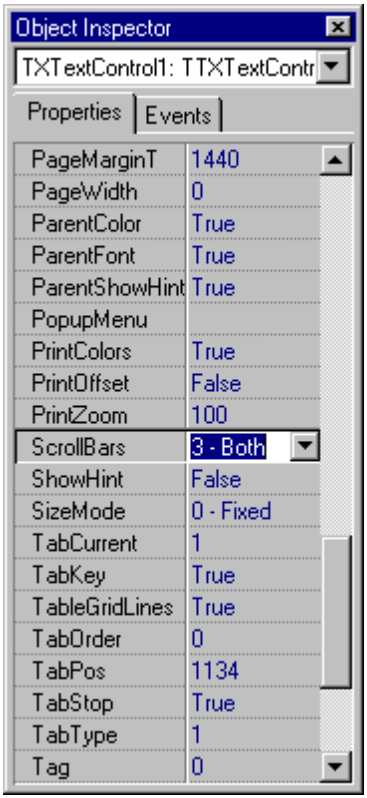
```
procedure TForm1.FormShow(Sender : TObject);
begin
   TXTextControl1.ButtonBarHandle := TXButtonBar1.hWnd;
   TXTextControl1.RulerHandle := TXRuler1.hWnd;
   TXTextControl1.StatusBarHandle := TXStatusBar1.hWnd;
end;
```

Running the Program

The text processor is not yet finished, but we can make a first attempt at running it and seeing what it can do. Click the Start button. You can type in some text, select it with the mouse, copy it to the clipboard (use the <CTRL>+<C> and <CTRL>+<V> keys as long as there is no menu), select a different font, set tabs and do lots of other things. All of

these features have been built into the Text Control and can be used with almost no programming effort.

You will have noticed, however, that some features are still missing. For instance, if you resize the main window, the controls keep their old sizes. There is no menu, and there are no scrollbars either. We will fix this in the coming chapters.



Adding Scrollbars

To add Scroll Bars, click on the Text Control window to have its property list displayed. Click on the Scrollbars property and enter *3 - Both*. Select the PageWidth property and enter 12000, which is about the width of a letter in twips, the currently selected measurement. Set PageHeight to 15000 for now.

Resizing the Controls

Two steps are involved in making the controls resize properly when the main window is resized:

- Set the **Align** property to *alTop* for the Button Bar, the Ruler and the Text Control. Set it to *alBottom* for the Status Bar. This will adjust everything except the height of the Text Control.
- Change to the events listing in the property window and double-click the **OnResize** event. The code window should show an empty procedure for the **Resize** event:

```
procedure TForm1.FormResize(Sender: TObject);
begin

end;
```

Extend it as follows:

```
procedure TForm1.FormResize(Sender: TObject);
begin
   TXTextControl1.Height := ClientHeight - TXRuler1.Height
      - TXStatusBar1.Height - TXButtonBar1.Height;
   TXTextControl1.Width := ClientWidth;
   TXRuler1.Width := ClientWidth;
end;
```

This line of code will cause the Text Control's height and width to be adjusted every time the size of the form is altered.

Adding a Menu

In this chapter, you will add a menu to the text processor to enable you to call the Text Control's built-in dialog boxes.

Use the Delphi Menu Component to create a *Format* menu with the items *Character...* and *Paragraph...*. (Please refer to the Delphi documentation if you need help with creating menus).

Add the following code to the Click procedures of the menu items:

```
procedure TForm1.Character1Click(Sender: TObject);begin
   TXTextControl1.FontDialog
end;
```

```
procedure TForm1.Paragraph1Click(Sender: TObject);begin
   TXTextControl1.ParagraphDialog;
end;
```

Start the program again. You should be able to use the menu items to call the font and paragraph dialog boxes.

Again use the *Menu Design Window* and create an *Edit* menu containing items for *Cut*, *Copy*, and *Paste*. The code for these menu items is:

```
procedure TForm1.Cut1Click(Sender: TObject);
begin
   TXTextControl1.Clip (1);
end;

procedure TForm1.Copy1Click(Sender: TObject);
begin
   TXTextControl1.Clip (2);
end;

procedure TForm1.Paste1Click(Sender: TObject);
begin
   TXTextControl1.Clip (3);
end;
```

After adding these menu items, you can exchange formatted text with other word processors via the clipboard.

The last menu for now shall be a simple file menu. Create a *File* menu including the items *Load...* and *Save As...*. Place a common dialog box icon on the form and enter the following code, which will call the common dialog box to get a file name from the user, and will then load respectively save the selected file:

```
procedure TForm1.Load1Click(Sender:
TObject);
const
   TXT_FIlE = 1;
   TXM_FILE = 3;
begin
   OpenDialog1.Title := 'Open file';
   OpenDialog1.Filename := '';
   OpenDialog1.Filter
      := 'Text Control Demo (*.txm)|*.txm
            |Plain text(*.txt)|*.txt';
   OpenDialog1.FilterIndex := 1;
   If OpenDialog1.Execute then begin;

   // Pass the filename to the text control
```

```
   If UpperCase(copy(OpenDialog1.Filename,
      length(OpenDialog1.filename)-2, 3)) = 'TXM' then begin
         TXTextControl1.Load(OpenDialog1.Filename,
                                  0, TXM_FILE, 0);
   end
   else
      TXTextControl1.Load(OpenDialog1.Filename,
                           0, TXT_FILE, 0);
   end;
end;

procedure TForm1.Saveas1Click(Sender: TObject);
const
   TXM_FILE = 3;
begin
   SaveDialog1.Title := 'Save as ...';
   SaveDialog1.Filename := '';
   SaveDialog1.Filter := 'Text Control Demo (*.txm)|*.txm';
   SaveDialog1.FilterIndex := 1;
   SaveDialog1.DefaultExt := 'txm';
   if SaveDialog1.Execute then begin;
      // Pass the filename to the text control
      TXTextControl1.Save(SaveDialog1.Filename,
                                  0, TXM_FILE, 0);
   end;
end;
```

What Comes Next

It goes, of course, without saying that Text Control has many more features than those included in our little demo program. It is up to you now to include zoom, images, tables, OLE objects, paragraph frames and whatever else makes up a full-blown word processor. If you need some hints about how to integrate special features, have a look at the source code of the other sample programs or post a message in the Text Control support forum at http://www.textcontrol.com.

Text Control Programming

This chapter is a guide to programming Text Control and its tools, explaining the parts which have been omitted from the *Creating a Simple Word Processor* example.

Working with Files

Text Control uses 5 different file formats:

- Its own native format, which you would normally use to store data in document files.
- The Rich Text Format (RTF), which can be used to exchange formatted text with other applications.
- HTML
- Microsoft Word format
- Unformatted text in ANSI or Unicode format.

An example of how to use the native file format has already been presented in the previous chapter. Using RTF, HTML, Word or unformatted text is just as simple: All you have to do is specify the format you want to use as a parameter of the **Load** or **Save** method.

Using RTF, HTML, Word and unformatted text you can only read or write the contents of a single Text Control from or to a file. Using the native file format, however, you can write a file header prior to saving the Text Control data, or even write the contents of several Text Controls to one file.

The *Forms1* sample program, which is described in the next but one section, shows you how to write the contents of multiple Text Controls to a single file. The *MDIDemo* sample shows you how to write a file header prior to the Text Control's data and how to use RTF, HTML, Word and unformatted text.

Printing

Delphi provides a printer object that can be used to print the contents of a Text Control.

The following example sends the contents of a Text Control, which can be several pages long, to the default printer:

```
procedure TForm1.Print1Click(Sender: TObject);
var wPages, No : Integer;
begin
   wPages := TXTextControl1.CurrentPages;
   Printer.BeginDoc;
   for No := 1 To wPages do begin
      TXTextControl1.PrintDevice := Printer.Canvas.Handle;
      TXTextControl1.PrintPage(No);
      if No <> wPages then
         Printer.NewPage;
   end;
   Printer.EndDoc;
end;
```

After storing the number of pages in a local variable called *wPages*, the printer object is initialized with the **Printer.BeginDoc** statement, The **For .. Do** loop runs from 1 to *wPages* to print all of the pages. Inside the loop there are three more lines of code which print a single page:

1. The device context handle of the printer object is assigned to Text Control's **PrintDevice** property. Without this step, a device context which is compatible to the screen device would be used, resulting in poor print quality.
2. The number of the page to be printed is passed as a parameter to the **PrintPage** method. This will also start the printing process.
3. The printer object's **NewPage** method is invoked to advance to the next page if there is one left.

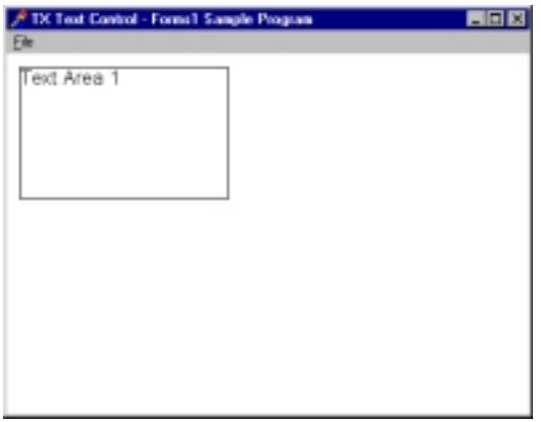
Everything else, like calculating the line and page breaks, is done internally by Text Control. The formatting is based on the values of two groups of properties:

- **PageHeight** and **PageWidth** determine the dimensions of the printed page.
- **PageMarginB**, **PageMarginL**, **PageMarginR** and **PageMarginT** determine the print margins.

These properties are normally set in a page setup dialog box.

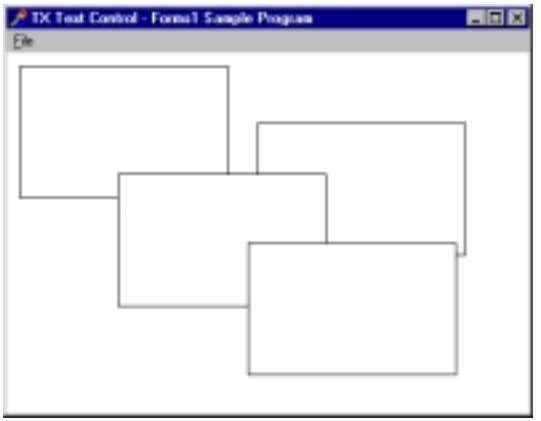
Using Multiple Controls

This chapter shows how to use Text Control in programs which have several text fields placed on a single page. Think of a program to print labels, to fill out forms, or to mask data entry. The *Forms1* sample program, which can be found in the *samples* subdirectory provides the basic functionality for applications of this kind.



Running the Sample Program

Initially, when the program is started, the main window contains one framed Text Control where text can be entered. The rest of the window is empty.



What you can do with the program is:

- Move the Text Control by pressing the ALT key and dragging the window with the mouse.

- Resize the Text Control by pressing the ALT key and dragging the window borders with the mouse.
- Create additional controls by clicking on an empty part of the main window.
- Save, load or print.

To keep things simple, there are no scrollbars in the main window and no menu items except the ones listed above. Scrollbars, zoom and a few other features will however be added in the next chapter.

How it Works

The *Forms1* sample uses a control array for the text fields. The first Text Control, the one which you see when you start the program, is placed on the form at design time. More controls are created when you click on an empty area of the form. These controls are created dynamically with the Delphi **Create** method. After creation initial values are assigned:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button:
TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
   MaxID := MaxID + 1;
   TX := TTXTextControl.Create(Form1);
   TX.Parent := Form1;
   TX.Top := Y;
   TX.Left := X;
   TX.Width := TXTextControl1.Width;
   TX.Height := TXTextControl1.Height;
   TX.Name := 'TXTextControl' + InttoStr(MaxID);
   TX.SizeMode := 3; // Move- and sizeable
   TX.BringtoFront;
   TX.OnMouseDown := TXTextControl1MouseDown;
end;
```

Clicking on an existing text field brings it to the front. This is done by changing the Z order when a Click Event has occured:

```
procedure TForm1.TXTextControl1MouseDown(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
   TTXTextControl(Sender).BringtoFront;
end;
```

The global variable *MaxID* counts the total number of controls; It is initialized to a value of 1 when the form is loaded.

Moving and resizing the controls is done by Text Control itself. To enable these functions, the **SizeMode** property must be set to *3 - Move and Sizeable*.

Saving the Controls

Saving a document which has been created with this program necessitates storing not only the data contained in the Text Controls, but also the number and the positions of the controls. In addition, a format identifier should be stored to enable the load routine of the program to determine if it can process a file which it is about to load. The code on the following page shows you how to save the document.

```
procedure TForm1.Save1Click(Sender: TObject);
var
   outFile : file of byte;
begin
   // Create an "Open File" dialog box
   SaveDialog1.Title := 'Save as';
   SaveDialog1.Filename := '';
   SaveDialog1.DefaultExt := 'txm';
   SaveDialog1.Filter := 'TX Form Demo (*.txf)|*.txf';
   SaveDialog1.FilterIndex := 1;

   if SaveDialog1.Execute then
      Filename := SaveDialog1.Filename
   else
      Exit;

   try
      begin
      // Open the selected file
```

```
      AssignFile(outFile, filename);
      rewrite(outFile);
      // Write file header
      FileID.lVersion := File_ID;
      BlockWrite(outFile, FileID, sizeOf(FileID));

      //Save properties of all text controls
      BlockWrite(outFile, MaxID, sizeof(MaxID));
      For i := 1 To MaxID do begin
         TXhWnd := FindComponent('TXTextControl'
            + Inttostr(i));
         with TXProp do begin
            xPos := TTXTextControl(TXhWnd).Left;
            yPos := TTXTextControl(TXhWnd).Top;
            xSize := TTXTextControl(TXhWnd).Width;
            ySize := TTXTextControl(TXhWnd).Height;
         end;
         BlockWrite(outFile, TXProp, sizeof(TXProp));
      end;
      closeFile(outFile);

      //Save contents of all text controls
      For i := 1 to MaxID do begin
         TXhWnd := FindComponent('TXTextControl'
            + InttoStr(i));
         TTXTextControl(TXhWnd).Save(FileName, -1, 3, False);
      end;
   end;
   Except
      MessageDlg('Error saving ' + filename,
                 mtError, [mbOK], 0);
   end;
end;
```

The **Load** routine first reads the format identifier and the number of controls. Then it creates the required number of controls, loads their contents and finally moves them to their correct position:

```
procedure TForm1.Load1Click(Sender: TObject);
var
```

```
   lFilePos : LongInt;
   InpFile    : file of byte;
   bOpen : Boolean;
begin
   // Create an "Open File" dialog box
   OpenFileDialog.Title := 'Open file';
   OpenFileDialog.Filename := '';
   OpenFileDialog.Filter := 'TX Form Demo (*.txf)|*.txf';
   OpenFileDialog.FilterIndex := 1;

   if OpenFileDialog.Execute then
      Filename := OpenFileDialog.Filename
   else Exit;

   try
      begin
         // Open the selected file
         bOpen := False;
         AssignFile(inpFile, filename);
         reset(inpFile);
         bOpen := True;

         // Read file header
         BlockRead(inpFile, FileID, sizeOf(FileID));
         If FileID.lVersion <> FILE_ID Then begin
            MessageDlg('Wrong filetype: ' + filename,
                       mtError,[mbOK],0);
            Exit;
         End;
         // Destroy existing controls
         If MaxID > 1 then begin
            For i := 2 To MaxID do begin
               TXhWnd := FindComponent('TXTextControl'
                  + Inttostr(i));
               TTXTextControl(TXhWnd).Free;
            end;
         end;
         //Create text controls and load their contents
         BlockRead(InpFile, MaxID, sizeof(MaxID));
```

```
            For i := 1 To MaxID do begin
               BlockRead(InpFile, TXProp, sizeof(TXProp));
               If i <> 1 Then begin
                  TX := TTXTextControl.Create(Form1);
                  TX.Parent := Form1;
                  TX.Width := TXTextControl1.Width;
                  TX.Height := TXTextControl1.Height;
                  TX.Name := 'TXTextControl' + InttoStr(i);
                  TX.SizeMode := 3;
                  TX.BringtoFront;
                  TX.OnMouseDown := TXTextControl1MouseDown;
               end;
               TXhWnd := FindComponent('TXTextControl'
                  + Inttostr(i));
               TTXTextControl(TXhWnd).Left := TXProp.xPos;
               TTXTextControl(TXhWnd).Top := TXProp.yPos;
               TTXTextControl(TXhWnd).Width := TXProp.xSize;
               TTXTextControl(TXhWnd).Height := TXProp.ySize;
               TTXTextControl(TXhWnd).Text := '';
            end;
            lFilePos := FilePos(InpFile);
            closeFile(InpFile);
            bOpen := False;

            For i := 1 To MaxID do begin
               TXhWnd := FindComponent('TXTextControl'
                  + Inttostr(i));
               lFilePos := TTXTextControl(TXhWnd).Load(FileName,
                  lFilePos, 3, False);
               TTXTextControl(TXhWnd).Visible := True;
            end;
         end;
      Except
         MessageDlg('Error opening ' + filename,
                    mtError, [mbOK], 0);
         if bOpen then begin
            closeFile(inpFile);
         end;
      end;
   end;
```

Printing Multiple Controls

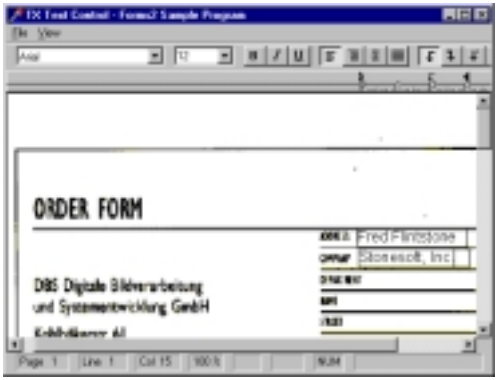
Printing a document is quite straightforward. The **PageWidth** and **PageHeight** properties are set to a value of 0 at design time, so the controls are printed like they are formatted on the screen. The print margin properties are used to specify the positions of the controls on the page.

```
procedure TForm1.Print1Click(Sender: TObject);
begin
   Printer.BeginDoc;
   For i := 1 To MaxID do begin
      TXhWnd := FindComponent('TXTextControl' + InttoStr(i));
      TTXTextControl(TXhWnd).PrintDevice := Printer.Handle;
      TTXTextControl(TXhWnd).PageMarginL
         := toTwip(TTXTextControl(TXhWnd).Left);
      TTXTextControl(TXhWnd).PageMarginT
         := toTwip(TTXTextControl(TXhWnd).Top);
      TTXTextControl(TXhWnd).PrintPage (1);
   end;
   Printer.NewPage;
   Printer.EndDoc;
end;
```

The complete source code of the *Forms1* sample program is contained in the *Forms1* sample source directory.

A Forms Filler

With the *Forms1* sample program, you can place text fields at arbitrary positions on a page. When you print the page, the text fields appear on the paper at exactly the same positions where they were previously placed on the screen. These features will be used in the following sample to create a program for filling out pre-



printed forms.
The scanned image of the form is shown in the background of the screen, enabling the user to easily determine the positions of the filled-out fields. He has only to click (with the CTRL key pressed) on the area of the form where he wants to put text and then start typing. The fields can be moved and resized afterwards by holding down the ALT key and dragging them with the mouse.

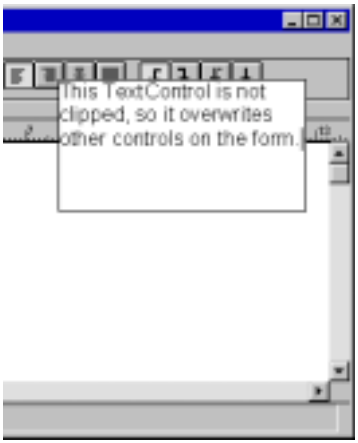
The source code for this example is contained in the *Forms2* sample source directory.

Adding ButtonBar, Ruler and StatusBar

The Button Bar, Ruler and Status Bar are used in a special way in this sample program. If you run the program and click on various fields you will notice that the tools automatically switch to the text field which has been clicked on. This switching is done internally by Text Control, so no programming is required for it. Each control is connected with the status bar, button bar and ruler after it is created.

Displaying the Background Image

The background image is displayed by an Image Control. This control is not a separate custom control, but a child window of the Text Control. To display the background image, create a Text Control which has the size of the whole page, and then load an image using Text Control's **ObjectInsertAsChar** method.



The Text Control which displays the background image has an additional function, which again saves a lot of programming work. It acts as a container for the Text Controls which are used as fill-out fields. (A container control enables you to draw other controls within it at design time. Examples of container controls are frames and picture boxes). The big advantage of a container is that it handles all of the clipping for the controls which have been created on top of it. Otherwise, scrolling the

background image would cause the text fields to overwrite anything that lies within the form's boundaries, like ButtonBar, Ruler, and even the scrollbars. It would require many calculations of field positions and sizes and some direct calls to the Windows DLLs on every scroll and resize event to do the clipping without a container control. Using the background Text Control as a container, you need only create the first text field inside of it, and everything else is done automatically.

Working with Transparent Text Controls

Run the program, load a background image and create a few text fields by clicking on this background image. You will notice that the text fields are transparent, so you can see the background image below. Using this feature in a program requires some fine-tuning of the clipping areas with the **ClipChildren** and **ClipSiblings** properties.

These two properties determine which areas of an image are repainted when a new part of a control becomes visible or when its contents have been changed.
For example, if one control is covered by another, it only has to be repainted if the one which lies on top of it is transparent. You will always want to repaint as little as possible to make the application run fast and to avoid unnecessary flickering on the screen. Furthermore you will not want your computer to spend time drawing things which are not visible.

For maximum flexibility in setting the clipping areas and mixing transparent and opaque controls, two properties have been implemented which share this task:

The **ClipChildren** property is used only for Text Controls which act as a container for other Text Controls. When **ClipChildren** is set to True, the areas occupied by the child controls are excluded from the update area. So, if as in the forms filler program, transparent controls are used as children of the container control, this property must be set to False.

The **ClipSiblings** property determines the behaviour between each of the child controls. It must be set to False if the program allows transparent Text Controls to overlap others.

Zooming

Zooming is simply done by setting the **ZoomFactor** property of each of the Text Controls:

```
procedure TForm1.N751Click(Sender: TObject);
begin
   Zoom := TXParent.ZoomFactor;
   TXParent.ZoomFactor := 75;
   For i := 1 to MaxID do begin
      TXhWnd := FindComponent('TXChild' + InttoStr(i));
      TTXTextControl(TXhWnd).ZoomFactor := 75;
   end;
   TXhWnd := FindComponent('N' + InttoStr(Zoom) + '1');
   TMenuItem(TXhWnd).Checked := False;
   N751.Checked := True;
end;
```

Using Marked Text Fields

Marked text fields are markers which are inserted in the text. They can be used to implement a wide range of special functions in a text processor. To name just a few:

- Mail Merge functions

- Spreadsheet-like calculation fields

- Bookmarks

- Automatic table of contents and index generation

- Hypertext viewers which include any kind of buttons, images, pop-up windows or even OLE objects in the text

Any group of characters within the text can be a marked text field. The maximum number of fields is 65,535. Text Control maintains the positions and numbers of the fields. It also takes care of loading, saving and clipboard operations.

A Simple Example

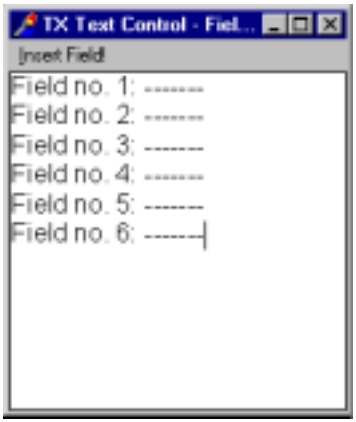
This first sample program will show you how fields are created and what happens when they are clicked on. The code shown here is contained in the *Field1* sample source directory.

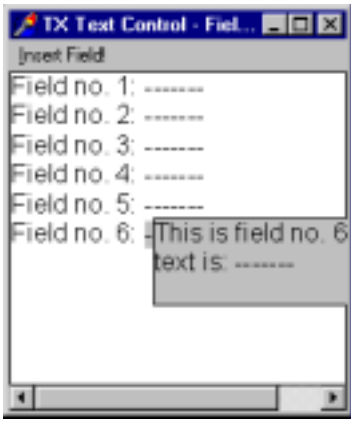
The program consists of a form with just one menu item, *Insert Field!*, with an exclamation mark to say that clicking on this item will cause an immediate action instead of dropping a menu. There are two Text Controls on the form, one of which is used as a normal text window (TXTextControl1), the other one as a pop-up window (TXTextControl2).

The following code is executed when the menu item is clicked on:

```
procedure TForm1.Insertfield1Click(Sender: TObject);
begin
   TXTextControl1.FieldInsert ('-------');
   TXTextControl1.FieldEditAttr
      [TXTextControl1.FieldCurrent] := $10 + $2 + $1;
end;
```

This inserts a field at the current caret position. If you move the cursor over the field, Text Control changes the mouse pointer to an upward pointing arrow (⇧) to indicate that there is something to click on.





If you click on the field, the application receives a **FieldClicked** event, to which it responds by popping up a window which displays the field number.

Only five lines of code are required for this:

```
procedure TForm1.TXTextControl1FieldClicked(Sender: TObject;
   FieldIndex: Smallint);
begin
   TXTextControl1.FieldCurrent := FieldIndex;
   TXTextControl2.Text := 'This is field no. '
      + InttoStr(FieldIndex) + '. Its text is: '
      + TXTextControl1.FieldText;
   TXTextControl2.Left :=
      toPixels(TXTextControl1.FieldPosX);
   TXTextControl2.Top := toPixels(TXTextControl1.FieldPosY);
   TXTextControl2.BringtoFront;
end;
```

The first line selects the marked text field which has been clicked on. Line 2 builds the string that is to be displayed in the pop-up window. Line 3 and 4 moves the pop-up window, which is initially hidden behind the text window, to the position of the marked text field. Line 5 puts the pop-up window in front of the text window to make it visible. When the mouse button is released, the text window is moved to the front again:

```
procedure TForm1.TXTextControl1MouseUp(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Inte-
ger);
begin
   TXTextControl1.BringtoFront;
end;
```

Bookmarks

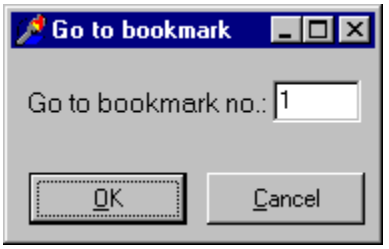
This example shows you how to use Text Control's marked text fields to create bookmarks. The first version will reference the bookmarks simply by their field numbers. The source code for this example is contained in the *Field2* sample source directory.

The sample application has a *Bookmark* menu with two items which are named *Insert* and *Go to...*. Clicking *Insert* creates a marked text field at

the current caret position. If a text selection exists, the selected text is converted into a field. If not, the character next to the caret is selected.

```
procedure TForm1.Insert1Click(Sender: TObject);
var
   fieldID : Integer;
begin
   If TXTextControl1.Text = '' Then
      Application.MessageBox ('Cannot insert a bookmark
         if the Text Control is empty.','ERROR',MB_OK)
   Else begin
      If TXTextControl1.SelLength = 0 Then
TXTextControl1.SelLength := 1;
      TXTextControl1.FieldInsert ('');
      fieldID := TXTextControl1.FieldCurrent;
      TXTextControl1.FieldEditAttr[fieldID] := $10 + $2 + $1;
   end;
end;
```

After typing in some text and inserting a few bookmarks, select the *Go To...* menu item. This will launch a dialog box which allows you to enter the number of the bookmark to jump to. There is no error processing in this example, so if you enter the number of a non-existent field, nothing will happen.



Clicking the 'OK' button executes the following procedure:

```
procedure TfrmGoto.Button1Click(Sender: TObject);
begin
   Form1.TXTextControl1.FieldCurrent
         := StrtoInt(Edit1.Text);
   Form1.TXTextControl1.SelStart
         := Form1.TXTextControl1.FieldStart - 1;
   Form1.TXTextControl1.SelLength
         := Form1.TXTextControl1.FieldEnd
            - Form1.TXTextControl1.FieldStart + 1;
   Close;
end;
```

The number which has been entered in the dialog box is taken as a value for the **FieldCurrent** property.

Adding Strings to Marked Text Fields

The source code for this example is contained in the *Field3* sample source directory.

In commercial word processors, bookmarks are normally referenced by names, not just by numbers. The names are typed in by the user when he creates a bookmark. The *Goto Bookmark* dialog box then presents a listbox or combobox in which one of the strings may be selected.

The *Insert Bookmark...* menu item in this version of the program creates a dialog box where the user can enter a label for the bookmark. When the 'OK' button is clicked, the following code is executed:

```
procedure TfrmInsert.btnOKClick(Sender: TObject);var fieldID
: Integer;
begin
   If Form1.TXTextControl1.SelLength = 0 Then begin
      If Form1.TXTextControl1.SelStart
         = Length(Form1.TXTextControl1.Text) Then
      begin
         Form1.TXTextControl1.SelText := ' ';
         Form1.TXTextControl1.SelStart
               := Form1.TXTextControl1.SelStart - 1;
      end;
      Form1.TXTextControl1.SelLength := 1;
   end;
   // Insert a field and store the bookmark name in its
   // FieldData property
   Form1.TXTextControl1.FieldInsert ('');
   fieldID := Form1.TXTextControl1.FieldCurrent;
   Form1.TXTextControl1.FieldEditAttr [fieldID]
         := $10 + $2 + $1;

Form1.TXTextControl1.FieldData[Form1.TXTextControl1.FieldCurrent]
```

```
      := Edit1.Text;
   Form1.TXTextControl1.SelLength := 0;
   Close;
end;
```

First, a marked text field is created at the current caret position. Second, the name of the bookmark, which is the text that has been typed in by the user, is stored in the **FieldData** property.

The *Goto Bookmark* dialog box contains a combo box which lists all of the bookmarks which have been created so far. The combo box is filled with the bookmark titles when its form is loaded:

```
procedure TfrmGoto.FormShow(Sender: TObject);
var
   nfieldID : Integer;
begin
   nFieldID := 0;
   cboBookmark.Clear;

   // Fill the combobox with bookmarks
   Repeat
      nFieldID := Form1.TXTextControl1.FieldNext(
                  nfieldID, 0);
      If nFieldID > 0 Then
         cboBookmark.Items.Add(
         Form1.TXTextControl1.FieldData[nFieldID])
   Until nFieldID = 0;

   // Copy first item to the edit control part
   // of the combo box
   cboBookmark.Text := cboBookmark.Items.Strings[0];
end;
```

When the ‘OK’ button is clicked, the bookmark list is searched for the string which has been selected in the combo box, and the corresponding marked text field is selected.

```
procedure TfrmGoto.Button1Click(Sender: TObject);
var
   nFieldID : Integer;
label
   Exit;
begin
   nFieldID := 0;
   // Search for the requested bookmark
   Repeat
      nFieldID := Form1.TXTextControl1.FieldNext(
                  nFieldID, 0);
      If nFieldID > 0 Then begin
         If Form1.TXTextControl1.FieldData[nFieldID] =
            (cboBookmark.Text) Then
         begin
            Goto Exit;
         end;
      end;
   until nFieldID = 0;

   Exit:
   // If the bookmark has been found, select it. Text
Control will then
   // automatically scroll to make it visible
   If nFieldID <> 0 Then begin
      Form1.TXTextControl1.FieldCurrent := nFieldID;
      Form1.TXTextControl1.SelStart
           := Form1.TXTextControl1.FieldStart - 1;
      Form1.TXTextControl1.SelLength
           := Form1.TXTextControl1.FieldEnd
              - Form1.TXTextControl1.FieldStart + 1;
      end
   Else
      Application.MessageBox(
                  'Bookmark not found.','ERROR',MB_OK);
   Close;
end;
```

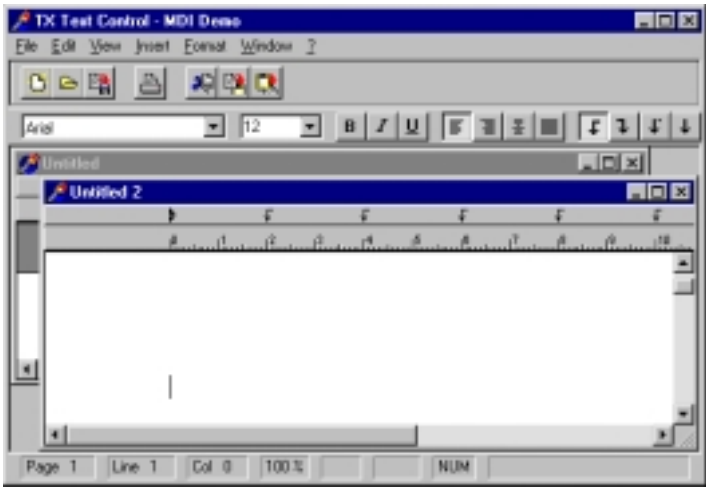
You can also extend the sample program with a dialog box, similar to the *Go To Bookmark...* dialog, in which a bookmark can be deleted

without deleting the text. This would require converting the marked text field to normal text. Use the **FieldDelete** method to achieve this.

More information about marked text fields and a list of all properties, methods and events that can be used with marked text fields, can be found in the Reference part, later on in this manual, in chapter "*Overviews - Marked Text Fields*".

A Word Processor

This chapter shows you how to use Text Control to write a standard word processor. The source code for this example is contained in the *MDIDemo* sample source directory.



Adding a PageSetup Dialog Box

The *Page Setup* dialog box is used to determine the page size and print margins. The maximum page size is restricted by the capabilities of the default printer. For implementation details, look at the source code of the *frmPageDlg* form.

A Print Dialog Box

When the *Print...* menu item is clicked, first a Common Dialog box is shown to let the user enter the range of pages, number of copies and printer specific information. The rest of the procedure, which is part of the MDIChild form, is just a loop which sets the appropriate Text Control properties for every page to be printed.

Search and Replace

Searching and replacing is entirely done in Text Control. You just have to assign a value of 1 for *Search* or 2 for *Search And Replace* to the **FindReplace** method. Text Control then opens the Windows Common Dialog box.

Using Paragraph Frames

With Text Control, you can add lines and frames to a paragraph or a range of paragraphs. For instance, you can put a line at the bottom of a caption like in the header of this manual.

The dialog box for paragraph frames is not included in the Text Control, but the source code is included in the MDI sample.

The properties which are responsible for paragraph frames are **FrameDistance**, **FrameLineWidth**, and **FrameStyle**.

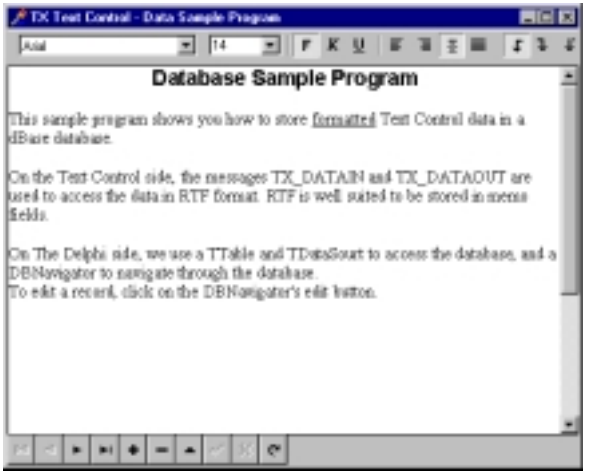
Dialog Boxes for Text and Background Color

This is also done with Common Dialogs. The color value returned from the dialog box is assigned to the **ForeColor** or **BackColor** properties.

Using Text Control with a Database

This chapter describes how to use Text Control to access databases with the Delphi TDBNavigator control. If you are not familiar with the TDBNavigator control please refer to the Delphi documentation.

The source code for this example is contained in the *Database* sample source directory.



The sample uses the TX_DATAIN and TX_DATAOUT messages to store data to the database and vice versa.Not only is the plain text stored, but also all formatting information, e.g. font and paragraph attributes, colors and image file names. The data is stored in RTF format, which is the easiest to handle.

The *Database* sample program is connected to a small data base which contains descriptions of some of Text Control's properties. You can browse through the records of the data base by clicking the TDBNavigator control buttons on the lower left side of the window. If you want to change something in the current record, press the button with the triangle.

Calling DLL Functions from Delphi Code

Sometimes it is necessary to access the Text Control DLL directly instead of using the properties and methods. There are messages which, because most users will never need them, have no corresponding properties, but which may be useful for your program.

The *CallDLLs* sample program, whose source code is contained in the *CallDLLs* sample source directory, shows you how to use these messages. You may want to browse through the DLL Reference online help file to see which other messages might be useful. The numbers of the Text Control messages are listed in *\samples\dll\inc\tx.h*.

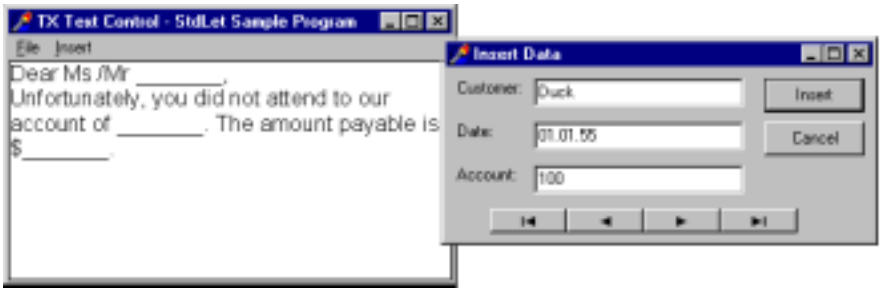
Mail Merge

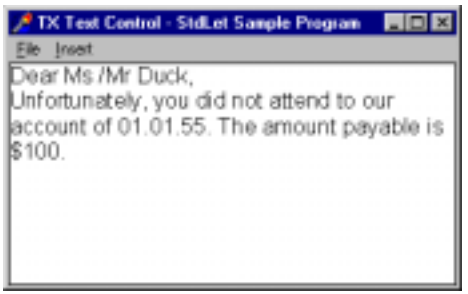
The chapter "*Using Text Control with a Database*" showed you how to store a Text Control's entire contents in a database field. For implementing functions like mail merge, however, the requirements are different: the contents of database fields have to be inserted at specified positions in a previously prepared document. The following sample program provides you with the basis of how to this.

The code shown here is contained in the *Stdlet* sample source directory.

The Sample Program

The program consists of two forms, Form1 for creating a text and Form2 for connecting it to the database.





Start the program and use the *File / Open...* command to load the sample file 'account.tx'. The file contains three fields which are to be replaced by database entries. Select *Insert / Data* to access Form2. When you click the *Insert* button in Form2, the contents of the three database

fields are copied to the text fields in Form1. You can select a different record by clicking one of the data control buttons in Form2, and then clicking *Insert* again to replace the fields.

How it Works

Each of the 3 edit controls on the second form is connected to a field in the database. The data is read from the database in the same way as in the *Database* sample.The only new thing is copying the data from the edit controls to the text fields in the document. This is done when you click on the *Insert* button:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
   Form1.TXTextControl1.FieldCurrent := 1;
   Form1.TXTextControl1.FieldText := Form2.Edit1.Text;
   Form1.TXTextControl1.FieldCurrent := 2;
   Form1.TXTextControl1.FieldText := Form2.Edit2.Text;
   Form1.TXTextControl1.FieldCurrent := 3;
   Form1.TXTextControl1.FieldText := Form2.Edit3.Text;
end;
```

To implement a real mail merge function you will have to add a dialog box in which the user can select the database to be used. You may also want to provide a variable number of database fields which are dependent on the contents of the selected database.

Using Hypertext Links

This chapter shows how to use Text Control's marked text fields to insert hypertext links and targets into text documents and how to respond to events which Text Control fires when the user clicks on a hypertext link.

The source code for the following examples is contained in the subfolders *Step1* to *Step4* of the *HyperLnk* sample source directory.

Step 1: Inserting a Hypertext Link

In this first sample program a hypertext link will be inserted in a text document. The document is saved then as a HTML file so that it can be viewed in a browser.

Hypertext links are handled as a special type of a marked text field. A hypertext link therefore is inserted by calling the **FieldInsert** method, and then specifying the type of the field with the **FieldType** property:

```
TXTextControl1.FieldInsert ('Text Control Web Site');
TXTextControl1.FieldType[TXTextControl1.FieldCurrent]
     := txFieldExternalLink;
```

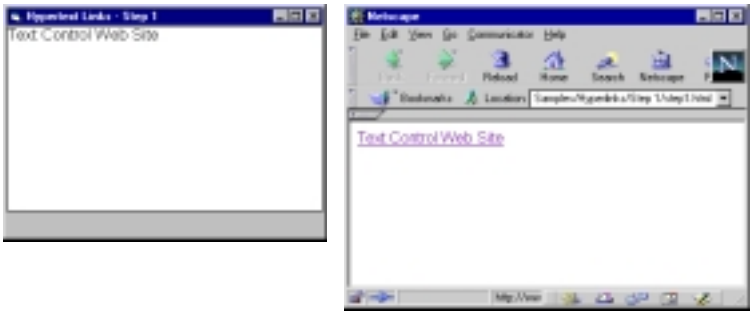
To store the target to where the link points, the **FieldTypeData** property is used:

```
TXTextControl1.FieldTypeData[TXTextControl1.FieldCurrent]
     := 'http://www.textcontrol.com';
```

The following line of code saves the document, containing the hypertext link, which has just been inserted as a HTML file in the sample folder:

```
TXTextControl1.Save (Application.GetNamePath + '.html',
        0, 4, False);
```

When this file is loaded with a web browser, the hypertext link will be displayed as specified in your browser's settings. Clicking on the link, will take you to the Text Control web site.



Note that there is no code for the **Click** events yet, so clicking on the hypertext link in the Text Control will have no effect. Also, the link is neither underlined nor colored.

Step 2: Adding a Dialog Box for Inserting Hypertext Links

In this second sample program a dialog box is created which enables the user, to insert hypertext links in a more convenient way. Additionally, hypertext links which have previously been inserted or loaded from a file, can be edited and modified. Note that, while hypertext links are usually associated with HTML files, they can as well be stored in RTF or Microsoft Word files, or in Text Control's proprietary format.

The dialog box has two text boxes. The first is for the text that represents the hypertext link in the document and the second is for the address, to where the link points. In the step 1 example, the representing text was "*Text Control Web Site*", and the address, to where the link points, was "*http://www.textcontrol.com*".

The same dialog box is used for both, inserting a new and editing an existing hypertext link. Depending on whether the current input position is inside of an existing link, this link is modified. Otherwise a new one is inserted.

The dialog form's property, *tx*, is used to pass a Text Control's reference and some information about how to display the hypertext links to the form.

```
procedure TForm1.HypertextLink2Click(Sender: TObject);
begin
   Form2.tx := TXTextControl1;
   Form2.ShowModal;
end;
```

When the form is loaded, the text boxes are filled with the text and link information when the current input position is inside of an existing link:

```
procedure TForm2.FormShow(Sender: TObject);
begin
// If the caret is inside an existing hyperlink,
```

```
// copy the hyperlink's text and link information to the
// text boxes on the form.

   If (Form1.TXTextControl1.FieldAtInputPos
        <> 0) Then  begin
      txtLinkedText.Text := Form1.TXTextControl1.FieldText;
      txtLinkTarget.Text
                 := Form1.TXTextControl1.FieldTypeData[
                    Form1.TXTextControl1.FieldCurrent];
   end Else begin
      txtLinkedText.Text := Form1.TXTextControl1.SelText;
      txtLinkTarget.Text := '';
   end;
end;
```

The user then can change the displayed information. The information is then transfered to the document by either inserting a link or modifying the existing one when the 'OK' button is pressed:

```
If tx.FieldAtInputPos <> 0 Then begin
   // editing an existing hyperlink
   tx.FieldText := txtLinkedText.Text;
   tx.FieldType[tx.FieldCurrent] := txFieldExternalLink;
   tx.FieldTypeData[tx.FieldCurrent] := txtLinkTarget.Text;
end Else begin
   // insert new hyperlink
   tx.FieldInsert (txtLinkedText.Text);
   tx.FieldType[tx.FieldCurrent] := txFieldExternalLink;
   tx.FieldTypeData[tx.FieldCurrent] := txtLinkTarget.Text;
   HighlightHyperlinks (tx, Form1.HypertextLinks1.Checked);
End;
```

Finally, there is a menu item to switch the character format of the hyperlink's text to blue colored and underlined style. The menu item calls the function **HighlightHyperlinks**, which is defined in the file Unit3.pas.

Step 3: Adding Targets

Step 1 and 2 only handle references to external resources, i.e. addresses of web pages or files. In this step, links to positions in the same document will be handled. These links are called internal links and the

positions, to where they point, are called targets. Targets are also refered to as anchors (in the context of HTML editors) or bookmarks (in word processors). When using this example, first add some text and then some targets with the *Insert / Target...* menu item. Finally use the *Insert / Hypertext Link...* menu item to add links to these targets.

Inserting a Target

Targets are realized again as a special type of a marked text field. The type and the target's name must be set with the **FieldType** and the **FieldTypeData** properties. Unlike links, targets have no visible text, therefore an empty field must be inserted with the **FieldInsert** method to insert a target:

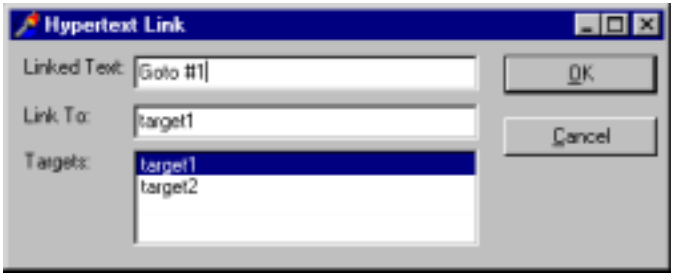
```
procedure TForm1.Target1Click(Sender: TObject);
var
   TargetName : String;
begin
   TargetName := InputBox('Target name:',
                 'Insert target', '');
   If TargetName <> '' Then begin
      TXTextControl1.FieldInsert ('');
      TXTextControl1.FieldType[TXTextControl1.FieldCurrent]
                 := txFieldLinkTarget;
      TXTextControl1.FieldTypeData[
              TXTextControl1.FieldCurrent] := TargetName;
   End;
end;
```

Only one text box is required to display the name of a target, so a simple InputBox statement can be used.

Inserting Links to Targets

To insert links to the just inserted targets, the *Hypertext Link* dialog box is extended with a list box showing the names of all targets the document contains. The **FieldNext** method is used to fill this list box:

```
Procedure TForm2.FillListboxWithTargets(Sender : TObject);
var
   FieldID : Integer;
begin
   ListBox1.clear;
   FieldID := tx.FieldNext(0, $100);
   While FieldID <> 0 do begin
      ListBox1.Items.Add(tx.FieldTypeData[FieldID]);
      FieldID := tx.FieldNext(FieldID, $100);
   end;
end;
```



When the user selects a target, the *Link To* field is filled with the target's name. After typing the link's text and pressing the 'OK' button, the link is inserted. An internal link is inserted in the same way as the external links from step 1, but the **FieldType** property now is set to **txInternalLink** and the **FieldTypeData** property is set to the target's name.

Jumping to a Target

After inserting internal links and targets, a jump must be realized. When the user clicks on a marked text field that represents a hypertext link, Text Control fires a **FieldLinkClicked** event. The information provided through this event can be used with the **FieldGoto** method to jump to the target:

```
procedure TForm1.TXTextControl1FieldLinkClicked(
     Sender: TObject; FieldId,
     FieldType: Smallint; var TypeData: WideString);
begin
   If FieldType = txFieldInternalLink Then
      TXTextControl1.FieldGoto(txFieldLinkTarget, TypeData);
```

```
end;
```

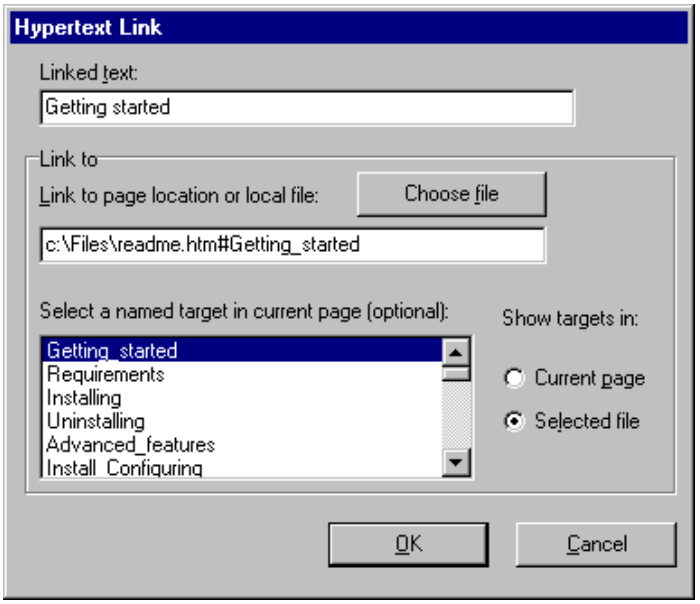
While the **FieldGoto** method is used for targets within the same file, links to external targets must be treated differently. When the **FieldLinkClicked** event occurs, and the **FieldType** parameter indicates that the link is external, then it depends on the type of the application, what to do. External links can point to, for instance, files on the local harddisk, or addresses in the internet.

Note that responding to the events is only required for making the hypertext links work while the text is edited in Text Control. If the text is saved to a file and displayed with a browser, then the hypertext links will work depending on the used browser.

Step 4: Adding Jumps to External Targets

Finally, in this step, jumps to other documents and jumps to targets in these documents are added.

An Enhanced Dialog Box for Displaying and Selecting Targets



Again the *Hypertext Link* dialog box is extended to choose an external file. A *Choose File...* button is placed on the form that triggers a common dialog. After the user has chosen a file, its name is displayed in the text box and the file is searched for internal targets:

```
procedure TForm2.CheckFileForTargets(tfile : String);
begin
   txhidden.LoadSaveAttribute[txEnableLinks] := True;
   txhidden.Load (tfile, 0, 4, false);
   FillListboxWithTargets (1);
   RadioButton2.Checked := True;
   loadedFile := tfile;
End;
```

For this purpose the file is loaded in a second, invisible Text Control. Then the **FieldNext** method is used as in step 3 to list all targets.

Jumping to an External Target

To implement the jump to an external link, the code added to the **FieldLinkClicked** event in step 3 must be extended. The following code does not handle jumps to internet addresses, it only implements jumps to targets in other files. To seperate a file from a name of a target, Text Control uses the '#' character. The following code separates the file name and the target's name, loads the file with the **Load** method and jumps to the target with the **FieldGoto** method:

```
Else begin
   // determine which type of link we have (see
   // List1_Click()) and remove the '#' charaacter.
   If (copy(txtLinkTarget.Text, 0, 1) = '#') Then begin
      LinkType := txFieldInternalLink;
      txtLinkTarget.Text := copy(txtLinkTarget.Text, 1,
         Length(txtLinkTarget.Text) - 1);
   end Else
      LinkType := txFieldExternalLink;

   If tx.FieldAtInputPos <> 0 Then begin
      // editing an existing hyperlink
      tx.FieldText := txtLinkedText.Text;
      tx.FieldType[tx.FieldCurrent] := LinkType;
      tx.FieldTypeData[tx.FieldCurrent]
                := txtLinkTarget.Text;
   end Else begin
      // insert new hyperlink
      tx.FieldInsert (txtLinkedText.Text);
```

```
      tx.FieldType[tx.FieldCurrent] := LinkType;
      tx.FieldTypeData[tx.FieldCurrent]
                  := txtLinkTarget.Text;
   End;
   HighlightHyperlinks (tx, Form1.HypertextLinks1.Checked);
   close;
End;
```

Loading and Saving Files containing Hypertext Links

When an HTML, RTF or Microsoft Word document is loaded, Text Control must convert containing hypertext links to appropriate marked text field, as described above. To perform this, a programmer must set the **LoadSaveAttribute**(**txEnableLinks**) before using the **Load** method. Otherwise hypertext links and target fields are not converted. When a document is saved, marked text fields that represent hypertext links, are always converted to the appropriate format.

If Text Control's proprietary format is used, setting **LoadSaveAttribute** is not necessary.

More information about hypertext links and a list of all properties, methods and events that can be used with marked text fields, can be found in the Reference part, later on in this manual, in the chapter "*Overviews - Marked Text Fields - Special Types of Marked Text Fields*".

Headers and Footers

This example shows how to use headers and footers. The source code is contained in the *Headers* sample source directory.

TX supports headers as well as footers. You also have the ability to create a different header or footer for the first page.

To insert a header or footer in the example, click on *Insert* and choose one of the four possible options. The code that is executed when clicking on one of the menu items is almost the same. For the *Header* menu item it looks as shown below. The line

```
TXTextControl1.HeaderFooter
             := TXTextControl1.HeaderFooter + txHeader;
```

informs Text Control that a header should be added to the current settings.

Setting the **HeaderFooterStyle** property to **txMouseClick** enables the user to activate the header with a single click rather than a double-click. Activating a header or footer with a double-click is Text Control's default setting. More information about how to use headers and footers and a list of all properties, methods and events that can be used with headers and footers, can be found in the Reference part, later on in this manual, in the chapter "*Overviews - Headers and Footers*".

When using properties, Text Control distinguishes between the main text and headers or footers. To switch between these different independent text parts, Text Control provides the **HeaderFooterSelect** method:

```
TXTextControl1.HeaderFooterSelect (txHeader);
TXTextControl1.SelText := 'Header';
TXTextControl1.HeaderFooterSelect (0);
```

This code selects the header, so that the following code affects the header and then sets the headers text. Finally the mode is reset to zero using the **HeaderFooterSelect** method. More information about programming with headers and footers see the chapter "*Overviews - Headers and Footers - Programming Headers and Footers*".

A header or footer is activated from programming code using the **HeaderFooterActivate** method. To delete a header or footer, simply substract the **txHeader** constant from the current **HeaderFooter** settings.

The following is the complete code of the menu item:

```
procedure TForm1.Header1Click(Sender: TObject);
begin
   If Header1.Checked = False Then begin
      TXTextControl1.HeaderFooter
         := TXTextControl1.HeaderFooter + txHeader;
      TXTextControl1.HeaderFooterSelect (txHeader);
      TXTextControl1.SelText := 'Header';
      TXTextControl1.HeaderFooterSelect(0);
      TXTextControl1.HeaderFooterActivate (txHeader);
```

```
      Header1.Checked := True;
   end Else begin
      TXTextControl1.HeaderFooter
                := TXTextControl1.HeaderFooter - txHeader;
      Header1.Checked := False;
   End;
end;
```

Drag and Drop

This example shows how to use the **InputPosFromPoint** method to realise a simple Drag&Drop in a Text Control application.

Drag&Drop in a text editor enables the user to drag a piece of text and drop it in a new location of the document. So, the incoming mouse events have to be analyzed and handled.

In the **MouseDown** event, the **InputPosFromPoint** method is used to get the character position the user has clicked on. The current input position and the length of the selection are stored in global variables, because they are needed in the **MouseUp** event. If the input position the user has clicked on, is inside of the current selection, dragging can be started. First a global variable named *dragging* is set to true and the **MousePointer** property is changed to indicate that dragging is in process. The text and format information of the current selection is copied to a memory buffer using the **SaveToMemory** method. Finally, the Text Control's **EditMode** property is set to *2 - read only*.

```
procedure TForm1.TXTextControl1MouseDown(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
   pos : LongInt;
begin
   //Get current input position and the current selection
   pos := TXTextControl1.InputPosFromPoint(
           toTwips(X), toTwips(Y));
   gblStart := TXTextControl1.SelStart;
   gblLength := TXTextControl1.SelLength;

   //Check if the click occured in the current selection
```

```
  If (gblStart <= pos)
        And (gblStart + gblLength > pos) Then begin
    //Start dragging
    data := TXTextControl1.SaveToMemory(3, True);
    dragging := true;
    Cursor := 2;
    TXTextControl1.EditMode := 2;
  End;
end;
```

In the **MouseUp** event procedure the **InputPosFromPoint** method is used again to get the character position where the user has left the mouse button. When dragging is in process and the input position is not inside the current selection, the drop opertation can be performed. The previously saved text now is inserted with the **LoadFromMemory** method after setting the new input position with the **SelStart** property.

```
procedure TForm1.TXTextControl1MouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  pos : LongInt;
begin

  pos := TXTextControl1.InputPosFromPoint(
        toTwips(X), toTwips(Y));

  If dragging Then begin
    //Check if the new input position is outside of
    //the current selection. If it's not, do not
    //copy the text
    If Not ((gblStart <= pos)
          And (gblStart + gblLength > pos)) Then begin
      TXTextControl1.SelText := '';
      If pos < gblStart Then
        TXTextControl1.SelStart := pos
      Else
        TXTextControl1.SelStart := pos - gblLength;

      TXTextControl1.LoadFromMemory(data, 3, True);
    End;
```

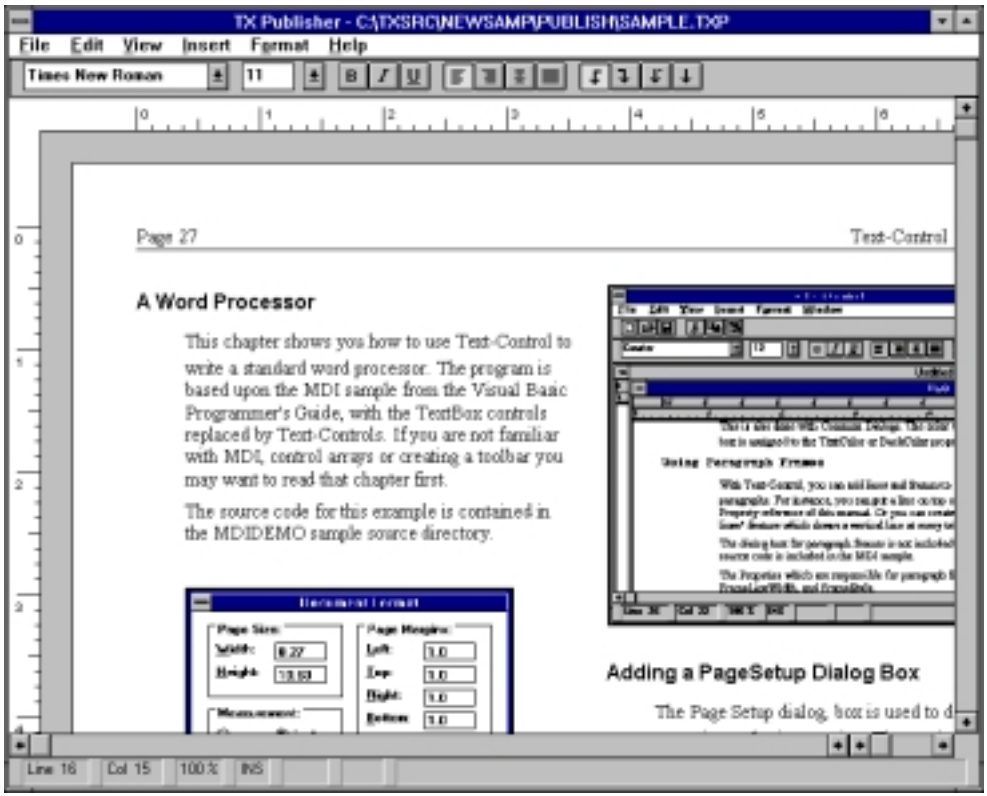
```
      //End dragging
      dragging := False;
      Cursor := 0;
      TXTextControl1.EditMode := 0;
    End;
end;
```

TX Publisher - An Advanced Example

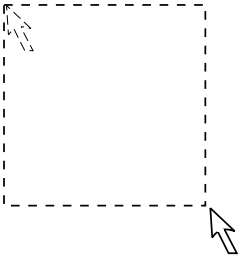
This sample progam is written entirely in Delphi, with no third party custom controls or DLLs except those included with Delphi itself. The program is intended to be used as a starting point for your applications, and it contains all the basic functions like loading and saving documents, printing, zooming, as well as the scroll interface. You can easily add more features and customize the program without having to start from scratch.

Text Frames and OLE Objects

TX Publisher works with text frames. This can entail pure text frames into which new text is entered or OLE objects. The type of frame is



defined in the Insert menu via the 'New Frame' menu item. In principle, the handling of text and OLE frames is the same. We will explain the frame handling using examples with text frames, and will then deal with the OLE object.



Drawing Text Frames

In order to draw a frame, click on an empty part of the page and, depressing the left mouse button, drag the mouse down and to the right. If you require a different page display for this, select it in the View menu via the 'Zoom' menu item.

The borders of the newly created text frame can be made visible by selecting 'Text Frames' in the 'View' menu. A paragraph ruler can be shown above the Text Frame. This setting is likewise made in the view menu, using the menu item ‘Paragraph Ruler’.

Text can now be entered into the newly created frame until it is full, at which point the frame has to be enlarged or the next frame has to be created. Alternatively you can draw all the required text frames successively, and then start entering text. Note that you can only start entering text in the first frame.

Connecting Text Frames

The text frames are linked automatically. This means that text automatically flows from the current frame into the next frame when the current frame is full.

If you click on an empty text frame which is further down the chain, then the cursor will stay in the last window which contains text.

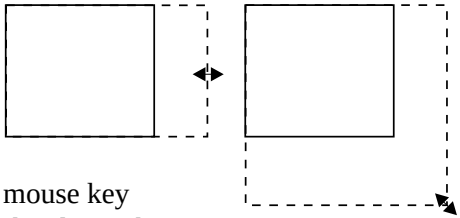
The text frames are numbered internally in their sequence of creation.

Deleting and Creating Frame Connections

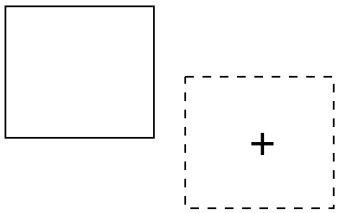
You can eliminate the connection between frames and, if required, regenerate them. You delete the connection to the following frame by clicking on the respective frame with the CTRL key depressed, and by answering the subsequent question displayed, ‘Delete connection to next window’, with Yes.

In order to create a connection, click on the frame to be connected to and keep the mouse button depressed until a symbol with a sheet of paper in a hand is displayed. Keeping the CTRL key and mouse button depressed, drag the symbol onto the frame to which you wish to create a connection. Answer the following question displayed, ‘Connect Frame No. x to Frame No. y’, with Yes. If it is not possible to create a connection, an error message will appear.

Changing Frame Size and Position



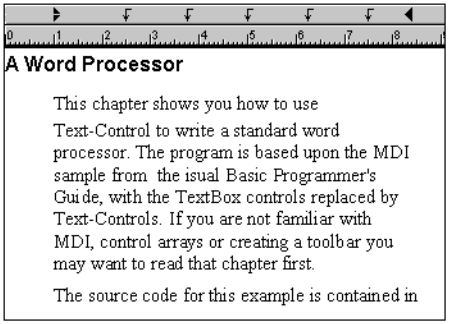
The size and position of a text frame can be changed subsequently. To change the size, click the frame borders or a corner of the frame with the ALT key depressed. Keep the mouse key depressed, and drag the respective border to the desired position.



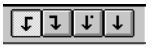
To change the position, click at any position within the frame with the ALT key depressed. Whilst keeping the mouse button depressed, drag the frame to the desired new position.

Setting Indents and Tabs

The currently active frame receives a paragraph ruler when this feature has been activated in the 'View' menu. Using the paragraph ruler you can set indents and tabs.



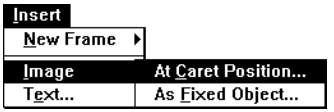
Indents can be changed by sliding the two small triangles on the left side of the ruler, and the large triangle on the right.



Tabs are left-aligned by default. To create right-aligned, decimal or centered tabs the tab type can be selected using the Button Bar.

You can set tabs by clicking at the desired position on the paragraph ruler. You can then shift the tab marker by clicking on it, and simultaneously dragging it along the ruler with the mouse button depressed. You can remove a previously set tab by pulling it downwards, away from the ruler. The maximum number of tabs is 14.

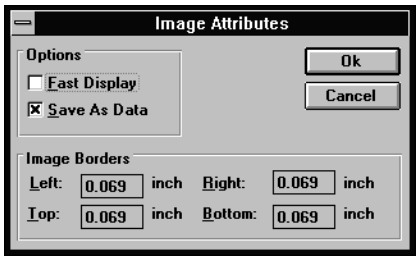
Using Images



Images can be inserted via the 'Insert/ Image' menu item or from the clipboard. The menu lets you choose between inserting the image 'At Caret Position' or 'As Fixed Object'. Images which are inserted at the caret position are treated like characters, and they move with the text as it changes.

When inserted as fixed objects, images have a fixed position on the page, and the text flows around them. The initial position of an image inserted in this way is one inch from the top left corner of the page. You can move it to the desired position just like you move text frames, which is by depressing the ALT key and dragging the image with the mouse.You can also change the size of the image in this way.

Clicking on an image and selecting 'Image...' from the 'Format' menu lets you select image attributes in a dialog box. You can adjust the size of the borders, i.e of a frame around the image where no text is displayed, and you can select if you want the image data to be included in your document file or if you just want to store a file reference. Storing the image data increases the size of your document file, but has the advantage of making the document independent of additional image files.

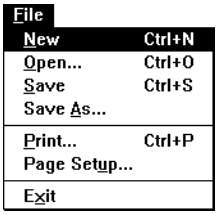


Images which are inserted from the clipboard are always inserted 'at caret position' and saved 'as data'.

OLE Objects

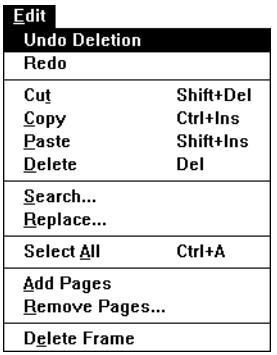
If you select 'OLE Object' in the 'Insert / New Frame' menu, frames are created in the following way. The frame is drawn as described above, by placing the mouse at the top-left corner of the frame to-be and dragging it down and towards the right. A dialog box entitled ‘Insert Object’ then appears. This dialog box also appears on the screen if you click over the frame with the right mouse button. You have the choice of creating a new object or of loading a file.

The File Menu



In the File menu you will find standard functions such as: New, Open, Save, Save As, Print, Page Setup, Exit. These will be familiar to you from various other Windows applications and will therefore not be described in more detail at this point.

The Edit Menu

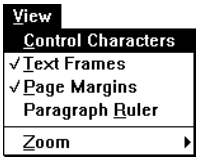


The Edit menu also includes a number of standard functions including Undo and Redo function, Cut, Copy, Paste, Delete, Search, Replace, and Select All. Regarding the Undo function, three different actions can be undone; Input, Deletion and Formatting.

Other menu items include ‘Add Pages’ and ‘Remove Pages’, with which you can insert and delete pages. When creating a document there are two document pages. In order to create additional pages, select Add Pages. Two further pages are then added to the existing ones. Using the small scroll bar at the bottom right, you can flick through the pages. You can delete the last two pages using 'Remove Pages'.

If you wish to delete a frame, initially activate it by clicking on it, and then select ‘Delete Frame’. After agreeing to ‘Delete Text Frame x’, the respective frame is removed.

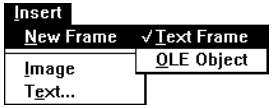
The View Menu



In the View menu you switch in or switch out one or more of the displays of Control Characters, Text Frames, Page Margins and Paragraph Rulers. You can also set the display size of the page view. Using ‘Zoom’ you have the following options available: Full Page, 30%, 50%, 75%, 100%, 200%.

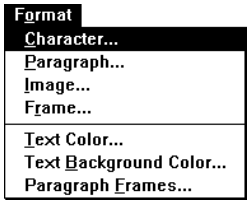
Regarding the Control Characters, soft and hard line breaks and blanks are displayed.

The Insert Menu



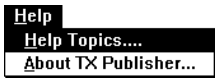
In the Insert menu you determine, via 'New Frame', the type of frame to be created. You can choose between 'Text Frame' and 'OLE Object'. For this purpose, read the previous pages. Using the ‘Image’ menu item, a picture can be imported, and by selecting ‘Text’, ASCII or RTF text files can be inserted at the current caret position.

The Format Menu



In the Format menu you can perform character and paragraph formatting. You can determine the text colour and text background colour, and using the menu item ‘Paragraph Frames’, you can define lines or frames for paragraphs.

The Help Menu



Using ‘Help Topics’ you call up the Online help service. You can also view an info window via the menu item ‘About TX Publisher’

How the Program Works

Much of the program's functionality is based on the concept of container controls. At the bottom of the control hierarchy there is a page ruler, which is placed directly on the form. On top of the page ruler there is a picture box which acts as a container for the document pages, which are themselves picture boxes. Finally, the text frames, OLE frames and the paragraph ruler use the page controls as containers. Although this may seem a bit complicated at first sight, it saves you a lot of programming work, because this approach helps to divide the program into logical blocks, and handles all the different clipping regions.You can see how the controls are put together when you look at the program in Delphi design mode. (See next page).

When the program is started, two document pages are created in a default size of A4 or Letter, depending on the system's country setting. The size of the workspace is then automatically adjusted so that the two pages can be shown side by side with a gray border around them. Settings which do not change during the program execution are made in the main form's OnCreate event, whereas settings which depend on the window size or the zoom factor are made in the form's OnResize event.

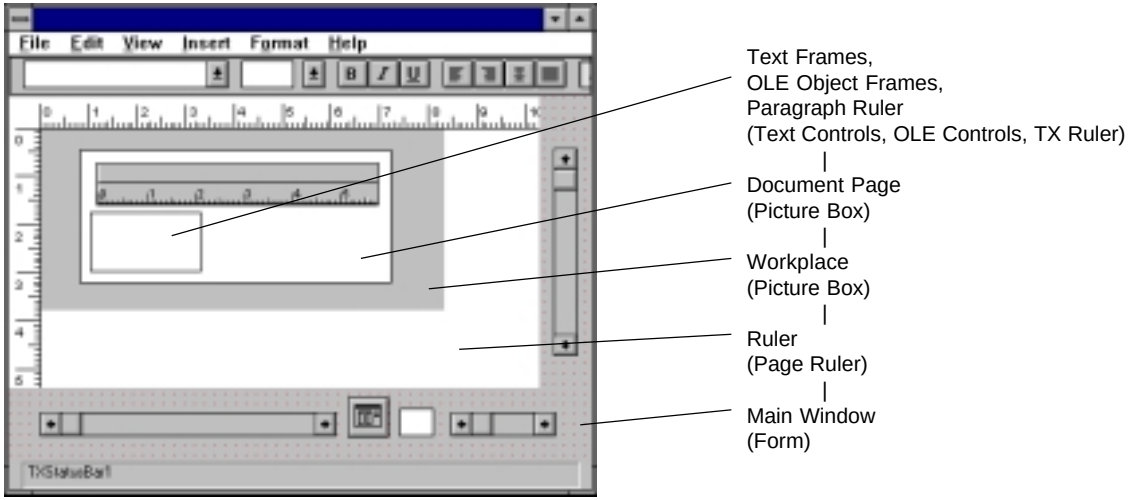
Managing Global Data

Most of the global data is managed by the controls themselves and thus does not have to be stored explicitly in variables.
For instance, the position and size of the text frames are stored in the control's Left, Top, Width and Height properties. Information which cannot be stored in control properties has been collected into a single global structure. This structure is called 'Doc' and contains information about zoom factor, page margins and the total number of text and OLE frames in the document.

Creating new Text and OLE Frames

A new frame is created when the user draws a rectangle on the page. This happens in three stages in response to the page control's mouse events:

On OnMouseDown, the mouse coordinates are stored as the top left



corner of the new control.

On OnMouseMove, a rectangle is drawn showing where the new control will be placed after the mouse button has been released.

Finally, when the OnMouseUp event occurs, the rectangle is deleted and a Text Control or OLE control is created at its coordinates.

The Text Controls and OLE controls, as well as the document pages, are implemented as control arrays, so a new instance of one of them can be created by calling the Load function. The newly created control is a child window of the sender control TXParentX:

```
TXNew.Parent := TTXTextControl(Sender);
```

Connecting Text Frames

The last step in creating a new text frame is to connect it to its predecessor so as to enable text to flow from one control to the next. This is simply done by assigning the window handle of the new Text Control to its predecessor's NextWindow property. The connection can be deleted later on by setting the property to a value of 0.

Deleting frames

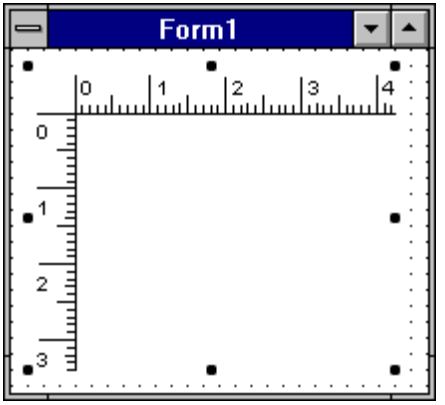
A frame is deleted when the user selects the 'Delete Frame' menu item.

This does not really remove it from the control array, but simply moves it into an unvisible TX Text Control called TXTrash. The controls in the trash are deleted automatically when the program is closed.

The Page Ruler Control

When you first start TX Publisher you will notice that the ruler looks different from the one in the standard version of TX Text Control. The ruler is in fact an additional custom control. Its filename is ‘PgRul.Ocx', which is short for ‘Page Ruler’.

The Page Ruler control can be used as a container for other controls. In the TX Publisher sample program, it is used as a document page, on which the text frames are placed. A detailed description of the Page Ruler's properties, methods and events can be found in the Reference part of this manual.



Other Languages

This chapter shows you how to use Text Control in programming environments other than Visual Basic or Delphi.

Standard C

If you are programming in an environment like Microsoft C 1.xx which does not support the OCX interface, you can use the Text Control DLL without the OCX wrapper. The interface is described in the DLL Reference.

Microsoft Visual C++ 4.x / 5.x / 6.x

Text Control can be used as an OCX with several Windows-based development environments. This chapter highlights procedures required to use Text Control as an OCX with the Microsoft Visual C++ environment.

Creating Applications in Visual C++

Before using Text Control with Visual C++, you should read the Microsoft Visual C++ 4.x / 5.x / 6.x documentation and on-line help.

Creating a Dialog, CFormView, or CView Based OCX Application

1. Start Visual C++.
2. From the File menu, choose New. The New dialog box appears
3. **VC 4.x:** In the New box, select Project Workspace and click OK.
   VC 5.x/6.x: In the New box, select Projects Tab.
4. The New Project Workspace dialog appears.
5. Browse to the desired directory path.
6. In the Name text box, type a name for your project. This will create a sub-directory of that name in the current path.
7. From the Type list, select MFC AppWizard(exe) to create a project based on the MFC library.

8. **VC 4.x**: Click the Create button.
   VC 5.x/6.x: Click the OK button.

The MFC AppWizard - Step 1 Dialog appears.

If you wish to create a Dialog based application, click the Dialog radio button, click NEXT and procede to the section, Dialog Based Applications. If you wish to create a CFormView based application, click the "Single Document" or "Multiple Documents" radio button, click NEXT and procede to the section, CFormView Based Application. If you wish to create a CView based application, click the "Single Document" or "Multiple Documents" radio button, click NEXT and procede to the section, CView Based Applications.

Dialog Based Applications

1. In the Step 2 dialog, click on the OLE Controls (**VC 5.x/6.x**: ActiveX Controls) check box to add built-in support for OCX products.
2. Click on NEXT button.
   The Step 3 dialog will appear.
3. In the Step 3 dialog, you can accept the default options by clicking the NEXT button.
4. In Step 4, you can accept the default options by clicking the FINISH button. VC++ will build your project.
   The New Project Information dialog will appear.
5. Click OK

CFormView Based Applications

1. In the Step 2 dialog you can accept the default options by clicking the NEXT button.
2. In the Step 3 dialog, click on the OLE Controls (**VC 5.x/6.x**: ActiveX Controls) check box to add built-in support for OCX products.
3. Click on Next button.
4. In the Step 4 and 5 dialogs you can accept the default options by clicking the NEXT button.
5. In the Step 6 dialog, select the class view name from the class list at

the top of the dialog.
CView will appear in the Base Class listbox.

6. In the Base Class listbox, change CView to CFormView.
7. Then click on the FINISH button to have VC++ build your project.

CView Based Applications

1. In the Step 2 dialog you can accept the default options by clicking the NEXT button.
2. In the Step 3 dialog, click on the OLE Controls (**VC 5.x**: ActiveX Controls) check box to add built-in support for OCX products.
3. Click on Next button.
4. In the Step 4 and 5 dialogs you can accept the default options by clicking the NEXT button.
5. In the Step 6 dialog, click on the FINISH button to have VC++ build your project.

Adding the Text Control Component to your Project

To insert a Text Control component into your project:

1. **VC 4.x**: From the Insert menu, choose Components.
   The Component Gallery dialog box appears.
   Select the OLE Controls tab.
   If the Text Control Text Control icon is not visible in the Gallery, click Customize to add the control.
   Select the control from the Component list on the right and click OK.
   This returns you to the Component Gallery.
   VC 5.x/6.x: From the Project / Add to Project menu choose Components and Controls.
   Open the Registered ActiveX Controls folder.
2. Select the Text Control icon in the Gallery and click Insert.
   The Confirm Classes dialog will display.
3. Click OK to confirm and exit the dialog.
4. Repeat steps 2 and 3 for the Status Bar, Ruler, and Button Bar controls.
5. Click Close to exit the Component Gallery.

The Text Control and its tools should now appear in the Control palette.

When VC++ adds components to your project, it creates CPP and H source files defining the class, properties, and methods for the control. It is a good idea to take a look at these files to understand what they contain. Methods and properties are not accessed the same in C++ as they are in many other languages like Visual Basic. When these files are generated, VC++ creates both a Get and Set function for most methods and properties. Text Control, for example, has a Text property. VC++ will create both a GetText and SetText member functions.

Adding the Component to your Dialog or CFormView:

1. In the Resource Editor, bring up the dialog that you want to place Text Control into.
2. Click on the Text Control component in the Editor's Control palette.
3. Draw the component on the dialog box.
4. Now this can be placed and sized as desired using the handles around the control.
5. Click on the right mouse button to bring up a floating menu. The design-time properties for the control can be viewed and modified through this menu.

Assigning Member Variables

Once you have added the text control to the dialog, it will be necessary to assign a member variable to each control to gain access to the methods and properties at runtime.

1. From the View menu, choose ClassWizard.
2. Select the Member Variables tab.
3. Select the control in the Control ID window for which you wish to add a variable and click the Add Variable button.
   The Add Member Variable dialog will display.
4. Type in the member variable name e.g. something like m_txctrl. Accept the default variable category and type, by clicking OK.
5. The MFC ClassWizard dialog will display the variable you added in the Control ID window.

6. Repeat steps 3 and 4 for each of the Text Control controls, specifying a new name for each.
7. Once you have added all the variables, click on OK in the MFC ClassWizard dialog to return to your project.

Adding the Text Control Component to your CView:

1. In the file list, bring up the header file for the view (<projname>view.h).
2. At the top of the file, include each of the Text Control control header files:

```
#include "tx4ole.h"
#include "txbbar.h"
#include "txruler.h"
#include "txsbar.h"
```

3. In the Attributes section, as a public member, add the following to create member variables for each of the controls in your view:

```
CTX4OLE  m_txctrl;
CTXBBAR  m_txbbar;
CTXRULER m_txruler;
CTXSBAR  m_txsbar;
```

4. Now through the file list, bring up the C++ source file for the view (<projname>view.cpp).
5. Start the ClassWizard. Make sure the view class is selected as the Class Name.
6. Select the View object in the Object Id listbox.
7. Select the "Create" message in the Messages listbox.
   The Create handler will initially come up with the following code:

```
return CWnd::Create(lpszClassName, lpszWindowName, dwStyle, rect,
pParentWnd, nID, pContext);
```

Change this to the following:

```
if (CWnd::Create(lpszClassName, lpszWindowName, dwStyle, rect,
                 pParentWnd, nID, pContext) == 0)
   return 0;
```

```
WCHAR szLic[] = L“AB-12345TS-1234567890“;
BSTR bstrKey = SysAllocString(szLic);
BOOL bSuccess = m_txctrl.Create(NULL, dwStyle, rect, this, 1000,
                                NULL, NULL, bstrKey);
SysFreeString(bstrKey);
if (!bSuccess)
   return 0;
if (m_txbbar.Create("TextControl ButtonBar", dwStyle, rect,
                  this, 1001) == 0)
   return 0;
if (m_txruler.Create("TextControl Ruler", dwStyle, rect,
                  this, 1002) == 0)
   return 0;
if (m_txsbar.Create("TextControl StatusBar", dwStyle, rect,
                  this, 1003) == 0)
   return 0;
return TRUE;
```

8. Start the ClassWizard. Select view class as the Class Name.
9. Select the View object in the Object Id listbox.
10. Select the "WM_SIZE" message in the Messages listbox.
11. Click on the Add Function button to create the OnSize handler function for this message.
12. Add the following code to the handler:

```
if (m_txctrl.m_hWnd && m_txbbar.m_hWnd && m_txruler.m_hWnd &&
m_txsbar.m_hWnd) {
   m_txctrl.MoveWindow(0, 60, cx, cy-(25+60));
   m_txbbar.MoveWindow(0, 0, cx, 30);
   m_txruler.MoveWindow(0, 30, cx, 30);
   m_txsbar.MoveWindow(0, cy-25, cx, 25);
}
```

Licensing the Control

The code added in the previous section, uses a license string to create a Text Control. Text Control is shipped as a CD version and as a trial version that can be downloaded and unlocked. The license string for the CD version users is the Text Control serial number. The license string for the trial version users is the customer key followed by the serial

number when the trial version is unlocked. When you use the locked trial version to test Text Control's features, use only your customer key as license string. In the code example above the customer key is "AB-12345" and the serial number is "TS-1234567890".

Connecting the Text Control Controls

Connecting the Controls:

1. In the Create handler, add the following code:

```
m_txctrl.SetButtonBarHandle(m_txbbar.GetHWnd());
m_txctrl.SetRulerHandle(m_txruler.GetHWnd());
m_txctrl.SetStatusBarHandle(m_txsbar.GetHWnd());
```

Handling Events in your Dialog or CFormView:

Assigning Message Handlers:

1. Start ClassWizard
2. In the Class Name listbox, select the Dialog or CFormView class that was created.
3. In the Messages listbox, select the desired message to handle and click on Add Function button to add a handler for this. For our example, select the "Click" event and click on the Add Function button to add the handler for this.
4. Click on the Edit Code button to edit the new function.
5. Add the following code in the function:

```
MessageBox ("Click Event","You clicked on the document");
```

6. Run the program and when the document is clicked on, the message "You click on the document".

Setting Properties in Visual C++

You can easily set specific properties for each of the controls you include in your project.

To set properties for a control:

1. Double-click on the control in your project that you wish to set properties for. The Control Properties dialog will display.

2. Select the appropriate tab for the property settings you wish to modify.
   Properties are grouped together in categories, such as paragraphs, fonts, and pages.
3. Modify the property settings as needed. For more information on each property, see 'Text Control Properties, Events, and Methods.'
4. Once you have set the properties for the active control, close the Control Properties dialog to return to your project.
5. Repeat steps 1 through 4 for each control.

Microsoft Access 2.0

The 16bit OCX can be used with Microsoft Access 2.0, but since Access went to manufacturing when the OLE control development kit was in an early prerelease state, some functions work different than in other programming environments.

1. You can't use Text Control (or any other OCX) as a bound control. The control must be inserted as an unbound object field, and you have to provide code for copying text from the data base to Text Control and vice versa.
2. Connecting the Text Control to the Ruler, ButtonBar and Status bar has to be done in 2 steps. (This example assumes that you have created a Text Control named 'tx' and a Ruler, ButtonBar and StatusBar named ' Ruler', 'ButtonBar' and 'StatusBar'.) In the FormLoad event, write:

```
Sub Form_Load ()
    Me!tx.object.RulerHandle = 1
    Me!tx.object.ButtonBarHandle = 1
    Me!tx.object.StatusBarHandle = 1
End Sub
```

This tells Text Control to send a ConnectTools event when all of the controls have been created ans are ready to be connected. In the ConnectTools event, write:

```
Sub TX_ConnectTools ()
    Me!tx.object.RulerHandle = Me!Ruler.object.hWnd
    Me!tx.object.ButtonBarHandle = Me!ButtonBar.object.hWnd
    Me!tx.object.StatusBarHandle = Me!StatusBar.object.hWnd
End Sub
```

Reference

Overviews

Text Formatting and Views

Text Control offers several ways, in which text may be formatted and viewed as described in the following list:

1. Control view:
The area for text formatting is the Text Control's control size. This is the default setting after creating a Text Control.

2. Control view with autoexpand:
The area for text formatting is the Text Control's control size, but this size is automatically expanded or reduced when the text exceeds it.

3. Control view and linked Text Controls:
The area for text formatting is the Text Control's control size, but text automatically flows to a following linked Text Control when it exceeds the control size.

4. Normal View:
The formatting width and height have been specified by the programmer and invisible text can be shown with a built-in scroll interface, with or without scroll-bars. The vertical scrolling amount depends on the text.

5. Page View:
The formatting width and height has been specified by the programmer. Text Control has a built-in scroll interface and displays pages with gaps, borders, margins and a gray background. The number of pages depends on the text.

6. Extended Page View:
This view works in the same way as the page view, but Text Control displays three-dimensional pages which are centered in the control's window.

Properties, Methods and Events

For text formatting, text view and scrolling the following properties, methods and events can be used:

Property/Method/Event	Description
AutoExpand Property	Sets or resets the autoexpand mode
AutoScroll Event	Occurs while expanding a selection with the mouse (control view only).
CaretOut Event	Occurs when the caret has been moved to a completely invisible control (control view only).
CaretOutBottom Event	Occurs when the caret has been moved down out of the control's visible area (control view only).
CaretOutLeft Event	Occurs when the caret has been moved to the left out of the control's visible area (control view only).
CaretOutRight Event	Occurs when the caret has been moved to the right out of the control's visible area (control view only).
CaretOutTop Event	Occurs when the caret has been moved up out of the control's visible area (control view only).
HExpand Event	Occurs when the Text Control has changed its control width (control view with autoexpand only).
HScroll Event	Occurs when the horizontal scroll position has been changed (normal view and page views only).
PageHeight Property	Gets or sets the formatting height.
PageMarginB Property	Gets or sets the bottom page margin.
PageMarginL Property	Gets or sets the left page margin.

PageMarginR Property	Gets or sets the right page margin.
PageMarginT Property	Gets or sets the bottom page margin.
PageWidth Property	Gets or sets the formatting width.
ScrollBars Property	Specifies which scrollbars are to be shown (normal and page views only).
ScrollPosX Property	Gets the current horizontal scroll position (normal and page views only).
ScrollPosY Property	Gets the current vertical scroll position.
VExpand Event	Occurs when the Text Control has changed its control height (control view with autoexpand only).
ViewMode Property	Specifies whether and how pages are to be displayed (normal and page views only).
VScroll Event	Occurs when the vertical scroll position has been changed (normal view and page views only).

Control View

The formatting area, which is the area Text Control uses to perform line breaks, is Text Control's control size. This means that every time the control is sized, the text is newly formatted.

A line break is automatically performed when the current input position reaches the control's right border. When the control's right border is reached without having a break character in a line, no line break is performed and Text Control indicates overflowing text with a vertical mark. This mark is displayed at the control's left or right border, depending on the paragraph alignment setting. When the number of lines exceed the control's height, overflowing text is indicated with a plus mark at the bottom of the control. The current input position,

indicated by the caret, can never leave the control. If the user tries to move the caret to overflowing text, the Text Control beeps.

When the Text Control is zoomed, the current control size is also zoomed to adapt the formatting area to the new zooming factor. Its position in relation to its container control or form is also zoomed.

Autoexpanding and control linking are only possible in this view. In autoexpanding mode the control's size is automatically expanded to the current text amount. This mode can be set with the **AutoExpand** property. After Text Control has expanded its control size it sends **HExpand** and/or **VExpand** events. A following linked Text Control can be set with **NextWindow** property.

Because the control view has no built-in scroll interface, Text Control has several events to enable a programmer to implement an external scroll interface. This can be important, for example when linked Text Controls are used to form a document with several pages. The **CaretOutBottom**, **CaretOutLeft**, **CaretOutRight** and **CaretOutTop** events are sent, when the current input position has been moved outside of the control's visible part. The **CaretOut** event is sent, when the current input position has been moved to a completely invisible control and the **AutoScroll** event is sent during extension of a text selection with the mouse. All of these events are only possible in the control view.

Normal View

With the **PageWidth** and **PageHeight** properties the formatting area can be changed by the programmer. This is necessary to realize a text editor that formats the text accordingly to a certain page width. The **ViewMode** property must be set to 0 for this view, which is the default value. In normal view the formatting height - the amount the user can scroll - depends on the current amount of text.

When the formatting width and/or height is greater than the current control size and the caret reaches the control's borders, scrolling is performed automatically to make the new input position visible. In addition, a horizontal and/or a vertical scroll-bar can be used to scroll to

text parts outside of the client area without changing the current input position. Scroll-bars can be set through the **ScrollBars** property.

If the Text Control is sized the text is not newly formatted and if the Text Control is zoomed the control's size and position are not changed. If the control is sized, visible scroll-bars are automatically hidden when they are no longer needed. Conversely, previously hidden scroll-bars are automatically shown when the control's size becomes smaller.

Page View

When the **ViewMode** property is set to 1, Text Control shows pages with margins and borders. The number of pages depends on the amount of text.

The **PageWidth** and **PageHeight** properties define the extension of the page including its margins. Page margins have a default value of one inch and can be changed through the **PageMarginB**, **PageMarginL**, **PageMarginR** and **PageMarginT** properties.

Extended Page View

When the **ViewMode** property is set to 2, pages are displayed in three dimensions with borders and shadows and they are centered in the control's window.

Like for many other extension settings, Text Control uses twentieths of a point to define the formatting area. This unit is often used in combination with text processing applications or fonts and is called TWIP. One TWIP is a 1/1440th of an inch.

In the normal view and in both page views Text Control has a built-in scroll interface. Text Control sends **HScroll** and **VScroll** events when it scrolls automatically. The current scroll positions can be obtained or set with the **ScrollPosX** and **ScrollPosY** properties.

Only with the page view and the extended page view can headers and footers be used. For more information how to handle headers and footers see "*Headers and Footers*".

Mixed Views

The control view and the normal view can be mixed, when either the **PageWidth** or **PageHeight** property is set to zero. For example, when specifying a width of zero and a height of non-zero, the text is formatted depending on the control's width but it is not limited to the control's height.

Headers and Footers

Properties, Methods and Events

With headers and footers the following properties, methods and events can be used:

Property/Method/Event	Description
HeaderFooter Property	Determines which headers and/or footers the document contains.
HeaderFooterActivate Method	Activates a certain header or footer.
HeaderFooterActivated Event	Occurs when a header or footer has been activated.
HeaderFooterDeactivated Event	Occurs when a header or footer has been deactivated.
HeaderFooterPosition Property	Specifies a header's or footer's position.
HeaderFooterSelect Method	Selects a certain header or footer to use a Text Control property for the header or footer instead for the main text.
HeaderFooterStyle Property	Determines style settings for headers and footers.

Using Headers and Footers

Headers and footers can only be used when the **PageWidth** and **PageHeight** properties have non-zero values. Headers and footers are only visible on the screen when the **ViewMode** property is set to 1 (page view) or 2 (extended page view).

Headers and footers are parts of a document. The **HeaderFooter** property determines whether headers and footers, only headers or only footers are contained. Additionally special headers and/or footers for the first page can be specified with this property. To edit a header or footer, it must be activated either with the **HeaderFooterActivate** method or with a mouse double-click in a header's or footer's area. An activated header or footer gets the input focus and its border is shown with a dotted frame. When a header or footer is activated, the main text is displayed in gray, otherwise a header's or footer's text is displayed in gray. Text Control fires **HeaderFooterActivated** and **HeaderFooterDeactivated** events to inform the application about activation or deactivation of headers or footers.

The **HeaderFooterStyle** property allows the following style settings:

1. Activation can be performed with mouse double-clicks and/or single mouse clicks.
2. The border of an activated header or footer can be solid, dotted or unframed.

The default style setting is a dotted frame and a mouse interface that activates a header or footer with double-clicks.

By default the top of a header has a distance of one centimeter from the top of the page and the bottom of a footer has a distance of one centimeter from the bottom of the page. With the **HeaderFooterPosition** property these values can be changed. The height of a header or footer depends on the header's or footer's current text.

When a document is loaded or converted from another format, contained headers and footers are automatically displayed. The **HeaderFooter** property returns the information which headers and/or footers the current document contains.

Programming Headers and Footers

Headers and footers are seperate text parts which are independent of the main text. When the user alters the text or the text format, for example with a connected button bar, Text Control uses the current input focus,

to determine whether the text format of a header, a footer or the main text is changed. The same occurs when the text is manipulated from programming code. For example when a table is inserted from a menu with the **TableInsert** method, the current input focus determines whether the table is inserted in a header's or footer's text or in the main text.

In addition to this default text part selection, a programmer can use the **HeaderFooterSelect** method to use a certain property with a certain text part. For example the following code alters the text of a header:

```
TXTextControl1.HeaderFooterSelect txHeader
TXTextControl1.Text = "This is the header's text"
TXTextControl1.HeaderFooterSelect 0
```

The first line selects the header, independent of the current input focus, the second line alters the text of the header and the third line returns to the default selection mode. There can be more than one property or method call between the two **HeaderFooterSelect** calls.

Almost all properties and methods can be used in this way with some exceptions. The following is a complete list of properties and methods which can be used with headers and footers:

Alignment
BaseLine
CurrentInputPosition
Fieldxxx (all field properties and methods)
Find
FindReplace
ForeColor
FormatSelection
Fontxxx (all font properties and methods)
FrameDistance
FrameLineWidth
FrameStyle
ImageDisplayMode
ImageFilename
ImageFilters

ImageSaveMode
Indentxxx
LineSpacing
LineSpacingT
Objectxxx (all object properties and methods)
ParagraphDialog
RTFSelText
SelLength
SelStart
SelText
TabCurrent
TabKey
TabPos
TabType
Tablexxx (all table properties and methods)
Text
TextBkColor

The following methods can only be used in conjunction with the **HeaderFooterSelect** method:

Load
LoadFromMemory
LoadSaveAttribute
Save
SaveToMemory

Tables

Properties, Methods and Events

With tables the following properties, methods and events can be used:

Property/Method/Event	Description
TableAtInputPos Property	Returns the identifier of the table with the current input position.
TableAttrDialog Method	Invokes a built-in dialog box for setting table attributes.

TableCanChangeAttr Property	Informs whether the attributes of a currently selected table can be changed.
TableCanDeleteLines Property	Informs whether the currently selected table rows can be deleted.
TableCanInsert Property	Informs whether a new table can be inserted at the current input position.
TableCellAttribute Property	Sets attributes of one or more table cells.
TableCellLength Property	Returns the number of characters in a table cell.
TableCellStart Property	Returns the character index of the first character in a table cell.
TableCellText Property	Gets or sets the text contents of a certain table cell.
TableColAtInputPos Property	Returns the number of the current input column in a table.
TableColumns Property	Gets a table's number of columns.
TableCreated Event	Occurs when a table has been created.
TableDeleted Event	Occurs when a table has been deleted.
TableDeleteLines Method	Deletes one or more complete rows of a table.
TableGridLines Property	Determines the visibility state of grid lines.
TableInsert Method	Inserts a new table.
TableNext Method	Can be used to enumerate all tables of a Text Control.
TableRowAtInputPos Property	Returns the number of the current input row in a table.
TableRows Property	Gets a table's number of rows.

Using Tables

Tables can be inserted into a Text Control either with the **TableInsert** method or via the **Load** method as part of a document formatted with the RTF or HTML formats. Text Control treats a table as a number of cells organized in rows and columns. Each cell can have as many lines and paragraphs as required. Paragraph formatting is performed in relation to a cell's borders. Each cell has a position and an extension in the document, within this area a cell's frames and text are drawn along with its paragraph and character formatting attributes. There can be a distance between the frame and the text.

Text can be selected either within a single cell or in steps of complete cells or rows. When a selection is deleted inside a table only the text is deleted. To delete one or more complete rows use the **TableDeleteLines** method. Tables can be copied to the clipboard and pasted from the clipboard. When a table is inserted at the first position of another table or immediately behind another table and both tables have the same number of columns they are combined into a single table. The insertion of one table inside another table is not possible.

A table's attributes are its frame width, distance between frame and formatted text, and background color. To alter the attributes of a table or part of a table, cells must be selected. Then a built-in dialog box can be opened with the **TableAttrDialog** method. When the selection extends over several tables or tables mixed with text, attributes cannot be changed. To get information about whether attributes can be changed or tables can be inserted or deleted, for example to implement a menu, the **TableCanChangeAttr**, **TableCanDeleteLines** and **TableCanInsert** properties can be used.

When the current input position is inside a table, the ruler shows the positions of all the cells in a table's row and the formatting attributes of the cell the input position belongs to.Then the cells' positions and extensions can be changed with a built-in mouse interface.

Programming with Table Identifiers

Each table can have an user-defined identifier which a programmer can set with the **TableInsert** method. Setting the identifier is not necessary but recommended when a table's text or attributes are to be changed from the program instead from an end-user. The user-defined identifier need not to be unique and remains valid if a table is saved and reloaded. When no identifier is set Text Control returns an own-selected one, which is unique and does not remain valid if a table is saved and reloaded.

When a table or a part of a table is inserted inside another table the inserted table becomes a part of the existing table and the inserted table's identifier is lost.

When a table with a user-defined identifier is inserted outside of all existing tables a new table is created and the table's identifier remains valid. Text Control informs the program with a **TableCreated** event that a new table has been created. The programmer can change the table identifier sent with the event by setting the *NewTableId* parameter of the event.

When a table is inserted from another application, which means it cannot have a user-defined identifier, Text Control sends an own-selected identifier with the **TableCreated** event and the program can change it.

When tables are imported with the **Load** method, **TableCreated** events occur only when text is inserted into an existing document or when an imported table has no user-defined identifier. Otherwise when a table with a user-defined identifier is saved and reloaded no event occurs.

When a table is completely deleted Text Control informs the program with a **TableDeleted** event.

The following properties and methods can be used with table identifiers to get information or to set table attributes regardless whether the current input position is or is not inside this table:

Property/Method	Description
TableCellText Property	Gets or sets the text contents of a certain table cell.
TableColumns Property	Gets a table's number of columns.
TableRows Property	Gets a table's number of rows.

When more than one table with a certain identifier exists, these properties and methods perform the operation with the original inserted table. In chains of linked windows these properties and methods can be used with any Text Control in the chain regardless of which control contains the table.

Marked Text Fields

A set of properties, methods and events has been implemented to define areas in the text of a Text Control called marked text fields. These fields can be used to create hypertext features like those in the Windows Help application, to realize database embedding while text of different datasets can be included into the text or to combine several fields with formulas as in spreadsheet applications.

Properties, Methods and Events

With marked text fields the following properties, methods and events can be used:

Property/Method/Event	Description
FieldAtInputPos Property	Returns the field identifier of the field containing the current input position.
FieldChangeable Property	Defines whether a field's text is or is not changeable.
FieldChanged Event	Occurs when the text of a field has been changed.
FieldClicked Event	Occurs when a field has been clicked.

FieldCreated Event	Occurs when a field has been created.
FieldCurrent Property	Specifies the current field.
FieldDblClicked Event	Occurs when a field has been double-clicked.
FieldData Property	Relates numeric or string data to a marked text field.
FieldDelete Method	Deletes a certain field.
FieldDeleteable Property	Defines whether a field is or is not deleteable.
FieldDeleted Event	Occurs when a field has been deleted.
FieldEditAttr Property	Defines field attributes for advanced editing.
FieldEnd Property	Returns a field's end position in the text.
FieldEntered Event	Occurs when the current input position has been moved from a position outside to a position inside a field.
FieldGoto Method	Sets the current input position to the beginning of a marked text field.
FieldInsert Method	Inserts a new field.
FieldLeft Event	Occurs when the current input position has been moved from a position inside to a position outside a field.
FieldLinkClicked Event	Occurs when a marked text field is clicked that represents the source of a hypertext link.
FieldNext Method	Finds the next marked text field.
FieldPosX Property	Returns a field's horizontal position.
FieldPosY Property	Returns a field's vertical position.

FieldSetCursor Event	Occurs when a cursor is moved over a field.
FieldStart Property	Returns a field's start position in the text.
FieldText Property	Gets or sets the text of a certain field.
FieldType Property	Sets or returns the type of a marked text field.
FieldTypeData Property	Sets or returns the data belonging to a marked text field of a special type.

Using Marked Text Fields

Fields can be inserted into a Text Control either with the **FieldInsert** method or via the **Load** method as a part of a document. The whole communication works with unique numbers returned by this method or defined by the user. To communicate with a field, the field must be previously set as the current field with the **FieldCurrent** property.

The current text can then be changed or retrieved with the **FieldText** property and a field can be deleted with the **FieldDelete** method. To get a field's position, either geometrically or as character position, the properties **FieldPosX**, **FieldPosY**, **FieldStart** and **FieldEnd** can be used. To get the number of the next field in the text or to enumerate all fields, the **FieldNext** method can be used.

Special attributes can be set with the **FieldChangeable** and **FieldDeleteable** properties. These attributes can prevent a field from beeing deleted or the text of a field from being changed. Further attributes which can help the end-user to edit the field's contents are described in the next chapter.

With different events Text Control informs the application about special conditions. The **FieldClicked** and **FieldDblClicked** event inform the application about mouse clicks, **FieldEntered** and **FieldLeft** indicate whether the current input position has been moved into or from a field. The **FieldSetCursor** event can be used to define the cursor when it is moved over a field. The default cursor is the up-arrow cursor. The

FieldChanged event occurs when the text of a field has been altered, and the **FieldDeleted** and **FieldCreated** event occur when fields have been deleted or created while inserting or deleting text with the keyboard or the clipboard.

Editing Marked Text Fields

When marked text fields are used in an editable Text Control and these fields are editable, the end-user can alter the contents of the field like any other text. Because it is not always unique whether the current input position is or is not inside a field some field attributes have been implemented to help the end-user to edit fields. These attributes can be used in any combination and must be set with the **FieldEditAttr** property.

When the current input position is in front or behind a field the next inserted character can either belong to the field or to the text outside the field. In normal editing mode an inserted character has the attributes of its preceding character which means that inserted text just behind a field belongs to the field and inserted text in front of a field does belong to the text in front of the field. To solve these problems extended editing features can be defined for every field with the **FieldEditAttr** property. It implements a second input position at the beginning and the end of the field. The end-user can switch between the two positions with the left and right arrow keys. This is especially important when a marked text field is at the beginning or the end of the complete text. For example when a field is at the end of the text the end-user can press CTRL+END to reach the text end. When this position is also the end of a marked text field the right arrow key can be pressed when the next inserted character should not belong to the field.

To help the end-user to find the correct position, additional settings can be performed which change the caret's width when it is inside a marked text field or display the complete text of a field with a gray background when the current input position is inside this field.

Each of the described attributes can be defined for a single field in any combination which means that different kinds of marked text fields can be implemented in a single Text Control.

Relating data to a marked text field

For each marked text field Text Control can store any data that can be set with the **FieldData** property. For example when a Text Control is used to show the contents of a database a marked text field can be created for each database field. The database's field names can then be related to the Text Control's marked text fields using the **FieldData** property.

Other parts of the program can use the **FieldData** property to retrieve the name of the database field to which a marked text field is linked. For example when the user has clicked on a marked text field, the **FieldData** property can be used with the field identifier, which has been specified through the **FieldClicked** event. The property then retrieves the name of the database field the user has clicked on.

The **FieldData** property accepts strings and numbers. When a marked text field is copied via the clipboard or saved to a file the data belonging to the field is also copied or saved. The usage of the **FieldData** property does not change the current text contents of a marked text field. When new data is set, all previously set data is overwritten independently of the kind of data involved.

Special Types of Marked Text Fields

Text Control supports special types of marked text fields that can be defined with the **FieldType** property. The following types are possible:

Type	Description
txFieldExternalLink	This field defines the source of a hypertext link to a location outside of the document.
txFieldInternalLink	This field defines the source of a hypertext link to a location in the same document.
txFieldLinkTarget	This field defines the target of a hypertext link.

txFieldPageNumber	This field displays the current page number. It can only be used in headers or footers.
txFieldHighlight	This field defines a piece of text that can be highlighted.
txFieldTopic	This field defines a position in a document that is the beginning of a topic.

All of these fields have the same general properties as standard marked text fields with the following exceptions: Fields of the type **txFieldLinkTarget** or **txFieldTopic** define text positions in a document. Therefore as they have no visible text, they cannot be edited and have no extended edit mode. Fields of the type **txFieldPageNumber** can only be used in headers or footers.

For each of the special field types Text Control handles some additional data, called type-related data. These data can be set or returned with the **FieldTypeData** property. For the types **txFieldExternalLink** and **txFieldInternalLink** these data are the information to where the link points. This can be an address or a file name and/or the name of a target in a document. Targets in documents can be realized with marked text fields, which have the type **txFieldLinkTarget**. These fields can have a name that is saved as type-related data. When the user clicks on a field of the type **txFieldExternalLink** or **txFieldInternalLink** a **FieldLinkClicked** event is fired including the information to where the link points. The **FieldGoto** method can be used to scroll to a target position and the **FieldNext** method can be used to enumerate all fields of a certain type.

To insert a field of a special type from programming code, use the **FieldInsert** method first and then set the type and its data. The following Basic example inserts a field that represents a link to the Text Control homepage:

```
Dim Field As Integer
TXTextControl1.FieldInsert "visit the Text Control homepage"
Field = TXTextControl1.FieldCurrent
TXTextControl1.FieldType(Field) = txFieldExternalLink
```

```
TXTextControl1.FieldTypeData(Field)
= "http://www.textcontrol.com"
```

When a user clicks on this marked text field, a **FieldLinkClicked** event is fired, containing the address of the homepage. To insert a field of the type **txFieldLinkTarget,** the created field must not have text. The following Basic example creates a field that represents such a target:

```
Dim Field As Integer
TXTextControl1.FieldInsert ""
Field = TXTextControl1.FieldCurrent
TXTextControl1.FieldType(Field) = txFieldLinkTarget
TXTextControl1.FieldTypeData(Field) = "first target"
```

This creates a field with the name "first target". The **FieldGoto** method can be used to scroll to this target:

```
TXTextControl1.FieldGoto txFieldLinkTarget, "first target"
```

When HTML, RTF or Word documents are loaded, source and target fields for hypertext links are automatically created. To perform this, set the **txEnableLinks** attribute with the **LoadSaveAttribute** property before using the **Load** method.

Fields of the type **txFieldPageNumber** display the current page number and can only be used in headers or footers. The following Basic example inserts a page number field:

```
TXTextControl1.FieldInsert ""
TXTextControl1.FieldType(TXTextControl1.FieldCurrent) =
txFieldPageNumber
```

Fields of the type **txFieldHighlight** can be used to mark pieces of text in a document that can be highlighted. This is useful, for instance, to highlight occurrences of a word found during a global search. The highlight color is stored as additional data for these fields. The **FieldGoto** method enables the programmer to scroll fom highlight to highlight. When RTF documents are loaded with the **Load** method and the **txEnableHighlights** attribute has been set previously with the **LoadSaveAttribute** property, all RTF '\cb*N*' keywords are automatically converted to fields of the type **txFieldHighlight**. *N* is the index of a color in the RTF color table.

Fields of the type **txFieldTopic** are text positions in a document defining the beginning of a topic. The **FieldGoto** method can be used to scroll to a topic with a certain number. When RTF documents are loaded with the **Load** method and the **txEnableTopics** attribute has been set previously with the **LoadSaveAttribute** property, all RTF '\sect' keywords are automatically converted to fields of the type **txFieldTopic** . These topics are numbered from 1 to n in the order they appear in the RTF document.

Resources

Text Control has several built-in resources like information strings, error messages and dialog boxes. These resources are available in different languages. When a new control is created Text Control selects the current set system language as the default one. With the **Language** property this setting can be altered independent of the system language. The description of the **Language** property lists all currently available built-in languages. To alter the language of the Button Bar and Status Bar the appropriate **Language** properties must be used.

To display resources in additional languages external resource libraries can be built and then set with the **ResourceFile** property. A resource library is a dynamic link library that only contains resources. The SAMPLES\TXRES subdiretory contains the basic files to create such a DLL file. See the chapter 1.15 "*Resources*" in the DLL Reference Manual for more information how to create a resource library.

To avoid conflicts with other programs that also uses own resources or with future versions of Text Control the following points are important:

1. The resource library should have a unique file name.

2. The resource library should be placed in the same directory as the final application. Get the full path name of the apllication's executable file at run time and specify the resource library's file name including this path when setting the **ResourceFile** property.

At runtime Text Control determines resources in the following way:

1.The Language property is initialized with the system default language. If the system language is not built-in, Text Control displays English resources.

2. When the **Language** property has been changed with an identifier of a built-in language, Text Control displays resources in this language independent of the system language.

3. When the **ResourceFile** property has been set, Text Control tries to load the resources from this library. In this case the **Language** property is ignored. When the resource library does not contain a needed resource or when the specified file could not be found, Text Control displays English resources without reporting an error.

4. Setting the **ResourceFile** and **Language** properties of a Text Control does not automatically set the appropriate properties of a connected Button Bar or Status Bar to the same values. These properties must be changed independently.

Text Control Data Types

The Text Control reference uses the following data types:

Data type	Description
Byte	Is a one-byte value with the range 0 to 255.
Boolean	Is a two-byte value that can be True or False.
Integer	Is a two-byte value with the range -32,768 to 32,767.
Long	Is a four-byte value with the range 2,147,483,648 to 2,147,483,647
Handle	32 bit: A four-byte value with the range 0 - 4,294,967,295. 16 bit: A two-byte value with the range 0 - 65536
String	Is a length-prefixed string of unlimited size.
Variant	Can be any of the priviously explained data types, including arrays of these types.

Text Control Properties, Events, and Methods

All of Text Control's properties, methods and events are listed in alphabetical order in the following table. A detailed description can be found in the following section.

Properties

Alignment
AutoExpand
BackColor
BackStyle
BaseLine
BorderStyle
ButtonBarHandle
CanRedo
CanUndo
ClipChildren
ClipSiblings
ControlChars
CurrentInputPosition
CurrentPages
DataText
DataTextFormat
EditMode
Enabled
FieldAtInputPos
FieldChangeable
FieldCurrent
FieldData
FieldDeleteable
FieldEditAttr
FieldEnd
FieldPosX
FieldPosY
FieldStart
FieldText
FieldType
FieldTypeData
FontBold
FontItalic
FontName
FontSize
FontStrikethru
FontUnderline
FontUnderlineStyle
ForeColor
FormatSelection
FrameDistance
FrameLineWidth
FrameStyle
HeaderFooter
HeaderFooterPosition
HeaderFooterStyle
HideSelection
hWnd
ImageDisplayMode
ImageFilename
ImageFilters
ImageSaveMode
IndentB
IndentFL
IndentL
IndentR
IndentT
InsertionMode
Language
LineSpacing
LineSpacingT
LoadSaveAttribute
MousePointer
NextWindow
ObjectCurrent
ObjectDistance
ObjectItem
ObjectScaleX
ObjectScaleY
ObjectSizeMode
ObjectTextflow
PageHeight
PageMarginB
PageMarginL
PageMarginR
PageMarginT
PageWidth
PrintColors
PrintDevice
PrintOffset
PrintZoom
ResourceFile
RTFSelText
RulerHandle
ScrollBars
ScrollPosX
ScrollPosY
SelLength

SelStart
SelText
SizeMode
StatusBarHandle
TabCurrent
TabKey
TableAtInputPos
TableCanChangeAttr
TableCanDeleteLines
TableCanInsert
TableCellAttribute
TableCellLength
TableCellStart
TableCellText
TableColAtInputPos
TableColumns
TableGridLines
TableRowAtInputPos
TableRows
TabPos
TabType
Text
TextBkColor
ViewMode
VTSpellDictionary
ZoomFactor

Methods

Clip
FieldDelete
FieldGoto
FieldInsert
FieldNext
Find
FindReplace
FontDialog
HeaderFooterActivate
HeaderFooterSelect
InputPosFromPoint
Load
LoadFromMemory
ObjectDelete
ObjectInsertAsChar
ObjectInsertFixed
ObjectNext
ParagraphDialog
PrintPage
Redo
Refresh
ResetContents
Save
SaveToMemory
TableAttrDialog
TableDeleteLines
TableInsert
TableNext
TextExport
TextImport
Undo
VTSpellCheck

Events

AutoLink
AutoScroll
CaretOut
CaretOutBottom
CaretOutLeft
CaretOutRight
CaretOutTop
Change
CharFormatChange
Click
ConnectTools
DblClick
Error
FieldChanged
FieldClicked
FieldCreated
FieldDblClicked
FieldDeleted
FieldEntered
FieldLeft
FieldLinkClicked
FieldSetCursor
HeaderFooterActivated
HeaderFooterDeactivated
HExpand
HScroll
KeyDown
KeyPress
KeyStateChange
KeyUp
MouseDown
MouseMove
MouseUp
Move
ObjectClicked
ObjectCreated
ObjectDblClicked
ObjectDeleted
ObjectGetData
ObjectGethWnd
ObjectGetZoom
ObjectMoved
ObjectPrint
ObjectScrollOut

ObjectSetData
ObjectSetZoom
ObjectSized
PageFormatChange
ParagraphChange
ParagraphFormatChange
PosChange
Size
TableCreated
TableDeleted
VExpand
VScroll
Zoomed

Alignment Property

Description: Returns or sets the text alignment for a Text Control.

Usage: TXTextControl.**Alignment** [= *value*]

The property's settings are:

Setting	Description
0 - Left aligned	(Default) Text is left-aligned.
1 - Right aligned	Text is right-aligned.
2 - Centered	Text is centered.
3 - Justified	Text is justified.
4	This value cannot be assigned to the property. Its purpose is to indicate that the selected text contains paragraphs which have different types of alignment.

Remarks: If the **FormatSelection** Property has previously been set to True, changing the **Alignment** Property affects only the currently selected paragaph. If **FormatSelection** has been set to False the setting applies to the entire control, in which case a value of 4 does not occur.

Data Type: Integer.

AutoExpand Property

Description: Specifies whether the control size should expand automatically when the text insertion or format changes results in text that does not fit into the Text Control anymore.

Usage: TXTextControl.**AutoExpand** [= *boolean*]

The property's settings are:

Setting	Description
True	The window size expands automatically.
False	Fixed window size.

This property is always set to False if the control is linked to other controls or if the **PageHeight** or **PageWidth** property is set to a value different from zero.

Data Type: Boolean.

Limitations: Runtime only.

AutoLink Event

Description: This event specifies that text will be inserted into the last control in a chain of linked windows. The program can avoid a text overflow at the end of the chain if it responds to this notification by an expansion of the chain. This event is sent before the text is inserted.

Syntax: **AutoLink**( )

See also: **NextWindow** Property.

AutoScroll Event

Description: This event occurs when the cursor leaves the visible portion of a Text Control's client area whilst a text selection is being expanded with the mouse. It is only sent if the cursor movement does not result in a caret movement. This happens if the cursor is moved outside the client area or if the cursor is moved over parts which are not covered with text below the last line. In all cases where the cursor movement results in a caret movement, the Text Control sends **CaretOutxxx** events.

Syntax: **AutoScroll**( )

See also: **CaretOut** Event, **CaretOutBottom** Event, **CaretOutLeft** Event, **CaretOutRight** Event, **CaretOutTop** Event.

BackColor Property

Description: Returns or sets the background color of a Text Control. Text Control uses the Microsoft Windows operating enviroment red-green-blue (RGB) color scheme.

Usage: TXTextControl.**BackColor** [= *value*]

The property's settings are:

Setting	Description
RGB colors	The valid range for a RGB color is 0 to &HFFFFFF. The high byte of a number in this range equals 0; the lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF).
System colors	Colors specified by the system color constants. If the high byte isn't 0, Text Control uses the system colors, as defined in the user's Control Panel settings.

Data Type: Long.

See also: **ForeColor** Property.

BackStyle Property

Description: Returns or sets a value indicating whether the background of a Text Control is transparent or opaque.

Usage: TXTextControl.**BackStyle**[= *value*]

The property's settings are:

Setting	Description
0 - Transparent	The Text Control has a transparent background.

1 - Opaque	(Default) The control's **BackColor** property setting fills the background.

Remarks: A transparent background is only possible when the control's container does not clip its controls. The most containers have a property to enable or disable clipping. For example a Visual Basic form has a **ClipControls** property.

Data Type: Integer.

See also: **BackColor** Property.

BaseLine Property

Description: Specifies the baseline alignment for selected text. A negative value is used to specify a subscript offset, a positive value for superscript. Text Control limits the baseline alignment to 960 twips in both directions.

Usage: TXTextControl.**BaseLine** [= *value*]

Data Type: Integer.

See also: **FormatSelection** Property.

BorderStyle Property

Description: Returns or sets the border style for a Text Control.

Usage: TXTextControl.**BorderStyle** [= *value*]

The property's settings are:

Setting	Description
0 - None	The Text Control has no border.
1 - Fixed Single	(Default) The Text Control has a fixed border.

Data Type: Integer.

ButtonBarHandle Property

Description: Specifies the button bar control to be used with a Text Control.

Usage: TXTextControl.**ButtonBarHandle** *[= ButtonBar.hWnd]*

Remarks: The Button Bar, like the Status Bar and the Ruler, is one of the additional controls which are contained in the Text Control OCX file.

Data Type: Handle.

Limitations: Runtime only.

See also: **RulerHandle** Property, **StatusBarHandle** Property.

CanRedo Property

Description: Informs whether an operation can be re-done using the **Redo** method.

Usage: TXTextControl.**CanRedo**

The property returns the following values:

Setting	Description
0	Nothing that can be restored.
10	The next redo operation restores inserted text.
11	The next redo operation deletes restored text.
12	The next redo operation restores the last formatting operation.

Data Type: Integer.

Limitations: Read only, Runtime only.

See also: **CanUndo** Property, **Undo** Method, **Redo** Method.

CanUndo Property

Description: Informs whether an operation can be undone using the **Undo** method.

Usage: TXTextControl.**CanUndo**

Remarks: The CanUndo Property has one of the following values:

Setting	Description
0	Nothing to be undone.
1	The next undo operation deletes inserted text.
2	The next undo operation inserts deleted text.
3	The next undo operation resets the last formatting operation.

Data Type: Integer.

Limitations: Read only, Runtime only.

See also: **CanRedo** Property, **Undo** Method, **Redo** Method.

CaretOut Event
CaretOutBottom Event
CaretOutLeft Event
CaretOutRight Event
CaretOutTop Event

Description: Occurs when the caret has been moved to a control that is completely out of the visible area.

Syntax: **CaretOut***xxx*( )

See also: **AutoScroll** Event.

Change Event

Description: Indicates that the contents of a Text Control have changed.

Syntax: **Change**( )

CharFormatChange Event

Description: Occurs when the formatting attributes of the selected characters have been changed. It also occurs if font settings have been changed because the Text Control has adapted fonts to a new output device.

Syntax: **CharFormatChange**( )

Click Event

Description: Occurs when the user presses and then releases a mouse button over a Text Control.

Syntax: **Click**( )

Clip Method

Description: Performs Text Control clipboard actions.

Usage: TXTextControl.**Clip** *Action*

The *Action* parameter can have one of the following values:

Value	Description
1	Cuts out the selected text and copies it to the clipboard.
2	Copies the selected text to the clipboard.
3	Pastes text from the clipboard.
4	Clears the selection.

Return Value: This method has no return value.

Data Types: *Action* Integer

Example: This Basic example copies the selected text from a Text Control named "TXTextControl1" to the clipboard when the user selects the "Edit/Copy" menu item:

```
Sub mnuEdit_Copy_Click ()
    TXTextControl1.Clip 2
```

```
End Sub
```

ClipChildren Property

Description: This property is only used for Text Controls which act as a container for other Text Controls or embedded objects. When this property is set to True, the areas occupied by the child controls are excluded from the update area.

Usage: TXTextControl.**ClipChildren** [= *boolean*]

The property's settings are:

Setting	Description
True	Exclude areas which are occupied by child controls from the update area.
False	(Default) Update areas which are occupied by child controls.

Data Type: Boolean.

See also: **ClipSiblings** Property.

Example: See Forms2 Basic sample program.

ClipSiblings Property

Description: This property determines the clipping behaviour of each of the child controls which belong to a common container control. It must be set to False if the program is to allow transparent Text Controls to overlap other Text Controls.

Usage: TXTextControl.**ClipSiblings** [= *boolean*]

The property's settings are:

Setting	Description
True	(Default) Excludes those areas occupied by other child controls from the update area.

False	Updates areas which are occupied by other child controls.

Data Type: Boolean.

See also: **ClipChildren** Property.

Example: See Forms2 Basic sample program.

ConnectTools Event

Description: MS Access only:
Occurs after all Text Controls, Rulers, ButtonBars and StatusBars have been created and are ready to be connected. This has to be done in 2 steps. (This example assumes that you have created a Text Control named 'tx' and a Ruler, ButtonBar and StatusBar named 'Ruler', 'ButtonBar' and 'StatusBar'.)

Example: In the FormLoad event, write:

```
Sub Form_Load ()
    Me!tx.object.RulerHandle = 1
    Me!tx.object.ButtonBarHandle = 1
    Me!tx.object.StatusBarHandle = 1
End Sub
```

This tells Text Control to send a **ConnectTools** event when all of the controls have been created and are ready to be connected. In the ConnectTools event, write:

```
Sub TX_ConnectTools ()
    Me!tx.object.RulerHandle = Me!Ruler.object.hWnd
    Me!tx.object.ButtonBarHandle = Me!ButtonBar.object.hWnd
    Me!tx.object.StatusBarHandle = Me!StatusBar.object.hWnd
End Sub
```

ControlChars Property

Description: Specifies if control characters are visible.

Usage: TXTextControl.**ControlChars** [= *boolean*]

The property's settings are:

Setting	Description
True	Control characters, like space or paragraph break, are visible.
False	Control characters are invisible.

Data Type: Boolean.

CurrentInputPosition Property

Description: Returns or sets an array of three values which specify the page, line and column number of the current text input position. These values are the same that are shown in Text Control's statusbar.

Usage: TXTextControl.**CurrentInputPosition** [= *Array*]

The array's values are:

Index	Description
0	Specifies the current page number. The first page has the number one.
1	Specifies the current line number. The first line has the number one.
2	Specifies the current column number. The first column has the number one.

Data Type: Array of 3 Long.

Limitations: Run time only.

CurrentPages Property

Description: Returns the number of pages contained in the current document.

Usage: TXTextControl.**CurrentPages**

Remarks: The value of this property depends on the size of the text as well as on the settings of the **PageHeight**, **PageWidth** and **PageMargin*x*** properties.

Data Type: Long.

Limitations: Read only, run time only.

See also: **PageHeight** Property, **PageWidth** Property, **PageMargin*x*** Properties, **PrintDevice** Property, **PrintPage** Method.

Example: See **PrintPage** Method example.

DataText Property

The **DataText** property is used internally by Visual Basic when Text Control is used as a bound control. This property is not to be manipulated by the developer and will likely be hidden in future releases.

DataTextFormat Property

Description: When using Text Control as a bound control, this property specifies if the data which is exchanged with a database is text or binary data.

Usage: TXTextControl.**DataFormat** *[= value]*

The property's settings are:

Setting	Description
0 - Text	Data is stored as text.
1 - Binary	Text and formatting information are stored in Text Control's own binary format.

Data Type: Integer.

DblClick Event

Description: Occurs when the user presses and releases a mouse button and then presses and releases it again over a Text Control.

Syntax: **DblClick**( )

EditMode Property

Description: Specifies whether the Text Control operates in edit mode or in one of the two read-only modes.

Usage: TXTextControl.**EditMode** *[= value]*

The property's settings are:

Setting	Description
0 - Edit	(Default) Edit mode. This mode can be used to edit and display text. The cursor is the text I-beam cursor.
1 - Read and Select	Read-only mode. This mode can be used to display and select text. The cursor is the standard arrow cursor.
2 - Read only	This mode can be used to display text only. Text input and selecting text with the mouse or the keyboard is not possible. The cursor is the standard arrow cursor.

Data Type: Integer.

Enabled Property

Description: Returns or sets a value that determines whether a Text Control can respond to user-generated events.

Usage: TXTextControl.**Enabled** *[= boolean]*

The property's settings are:

Setting	Description
True	(Default) Allows a Text Control to respond to events.
False	Prevents a Text Control from responding to events.

Data Type: Boolean.

Error Event

Description: Occurs when the Text Control reports an error.

Syntax: **Error**(*Number*, *Description*, *Scode*, *Source*, *HelpFile*, *HelpContext*, *CancelDisplay*)

The event procedure's parameters are:

Parameter	Description
Number	Is the error number.
Description	Is a corresponding error string. This string can be changed.
Scode	Is the OLE Status Code.
Source	Is the name of the module which caused the error.
HelpFile	Is the name of a help file.
HelpContext	Is the help context ID in this help file.
CancelDisplay	Can be set to True if the application wants to display its own error string. When this parameter is not set to True, the control will display a message box showing the error string.

Data Types:

Number: Integer
Description: String
Scode: Long
Source: String
HelpFile: String

HelpContext: Long
CancelDisplay: Boolean

FieldAtInputPos Property

Description: Returns the field identifier of the field containing the input position. Zero is returned when the input position is not inside a field.

Usage: TXTextControl.**FieldAtInputPos**

Data Type: Integer.

Limitations: Read only, run time only.

FieldChangeable Property

Description: Specifies if the contents of a marked text field can be changed by the user. The field identifier must have previously been determined with the **FieldCurrent** property.

Usage: TXTextControl.**FieldChangeable** [= *boolean*]

The property's settings are:

Setting	Description
True	The text which is contained in the field can be changed.
False	The text cannot be changed.

Data Type: Boolean.

Limitations: Runtime only.

See also: **FieldDeleteable** Property.

FieldChanged Event

Description: Occurs when the text of a marked text field has been changed.

Syntax: **FieldChanged**(*FieldId*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field that has been changed.

Remarks: The value of the **FieldCurrent** property is updated with the value given through the *FieldId* parameter.

Data Types: *FieldId* Integer

See also: **FieldClicked** Event, **FieldCreated** Event, **FieldDblClicked** Event, **FieldDeleted** Event, **FieldSetCursor** Event.

FieldClicked Event

Description: Occurs when a marked text field has been clicked on.

Syntax: **FieldClicked**(*FieldId*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field that has been clicked on.

Remarks: The value of the **FieldCurrent** property is updated with the value given through the *FieldId* parameter.

Data Types: *FieldId* Integer

See also: **FieldChanged** Event, **FieldCreated** Event, **FieldDblClicked** Event, **FieldDeleted** Event, **FieldSetCursor** Event.

FieldCreated Event

Description: Occurs when a marked text field has been pasted from the clipboard.

Syntax: **FieldCreated**(*FieldId*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field that has been created.

Remarks: The value of the **FieldCurrent** property is updated with the value given through the *FieldId* parameter.

Data Types: *FieldId* Integer

See also: **FieldChanged** Event, **FieldClicked** Event, **FieldDblClicked** Event, **FieldDeleted** Event, **FieldSetCursor** Event.

FieldCurrent Property

Description: Returns or sets the identifier of the current marked text field for the **Fieldxxx** properties, methods and events.

Usage: TXTextControl.**FieldCurrent** [= *FieldId*]

Data Type: Integer.

Limitations: Run time only.

Example: The Basic example creates a marked text field with a text content of 'New Field' and afterwards changes the text to 'Hello':

```
Sub Create()
   Dim FieldId As Integer

   'Create a marked text field and store its number
   TXTextControl.FieldInsert "New Field"
   FieldId = TXTextControl.FieldCurrent
   ..
   'Change the text
   TXTextControl.FieldCurrent = FieldId
   TXTextControl.FieldText = "Hello"
End Sub
```

FieldData property

Description: This property relates numeric or string data to a marked text field.

Usage: TXTextControl.**FieldData**(*FieldId*) [= *Data*]

The property's parameters are:

Parameter	Description
FieldId	Identifies the field that is to be manipulated.

Remarks: The specified data can be a long value or a character string. A long value of zero or an empty string deletes all data previously related to the specified marked text field.

Data Type: Long or String.

Limitations: Run time only.

See also: **FieldInsert** Method.

FieldDblClicked Event

Description: Occurs when a marked text field has been double-clicked on.

Syntax: **FieldDblClicked**(*FieldId*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field that has been double-clicked on.

Remarks: The value of the **FieldCurrent** property is updated with the number given through the *FieldId* parameter.

Data Types: *FieldId* Integer

See also: **FieldChanged** Event, **FieldClicked** Event, **FieldCreated** Event, **FieldDeleted** Event, **FieldSetCursor** Event.

FieldDelete Method

Description: Deletes the marked text field specified by the **FieldCurrent** property, or changes it to simple text.

Usage: TXTextControl.**FieldDelete** *DeleteTotal*

The *DeleteTotal* parameter can have one of the following values:

Value	Description
True	The marked text field is completely deleted.
False	The marked text field is deleted, but its text contents are preserved.

Return Value: The method returns True when the operation could be performed, otherwise it returns False.

Data Types: *DeleteTotal*: Boolean
Return value: Boolean

FieldDeleteable Property

Description: Specifies whether a marked text field can be deleted by the user. The field identifier must have previously been determined with the **FieldCurrent** property.

Usage: TXTextControl.**FieldDeleteable** [= *boolean*]

The property's settings are:

Setting	Description
True	The field can be deleted.
False	The field cannot be deleted.

Data Type: Boolean.

Limitations: Run time only.

See also: **FieldChangeable** Property.

FieldDeleted Event

Description: Occurs when a marked text field has been deleted.

Syntax: **FieldDeleted**(*FieldId*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field that has been deleted.

Remarks: The value of the **FieldCurrent** property is set to zero.

Data Types: *FieldId* Integer

See also: **FieldChanged** Event, **FieldClicked** Event, **FieldCreated** Event, **FieldDblClicked** Event, **FieldSetCursor** Event.

FieldEditAttr Property

Description: This property returns or sets attributes for advanced editing inside marked text fields.

Usage: TXTextControl.**FieldEditAttr**(*FieldId*) [= *Attr*]

The property's parameters are:

Parameter	Description
FieldId	Identifies the field that is to be manipulated.

The property's settings are:

Setting	Description
&H1&	Implements a second character input position at the beginning and the end of the specified marked text field.
&H2&	Performs normal editing at the beginning and the end of the specified marked text field.
&H4&	Changes the width of the caret when the character input position is inside the specified marked text field.
&H8&	Uses the normal text caret when the character input position is inside the specified marked text field.

&H10&	Displays the text of the specified marked text field with a gray background when the input position is inside this field.
&H20&	Displays the text of the specified marked text field with the standard control background when the input position is inside this field.
&H40&	Enables the normal double-click processing inside marked text fields that starts a wordwise selection.
&H80&	Disables the normal double-click processing inside marked text fields.

These values can be combined by adding the desired constant values. Changing one option does not affect the other. The default value of a newly inserted field is:

&H2& + &H8& + &H20& + &H80&

Data Type: Integer.

Limitations: Run time only.

See also: **FieldInsert** Method.

FieldEnd Property

Description: Returns the end position of a marked text field. The field identifier must have previously been determined with the **FieldCurrent** property.

Usage: TXTextControl.**FieldEnd**

Data Type: Long.

Limitations: Read only, run time only.

See also: **FieldStart** Property.

FieldEntered Event

Description: Occurs when the current input position, indicated by the caret, has been moved to a position that belongs to a marked text field. It only occurs if the caret has been moved using the keyboard. If the caret has been moved with a mouse click a **FieldClicked** event is sent.

Syntax: **FieldEntered**(*FieldId*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field that has been entered.

Remarks: This event that does not change the value of the **FieldCurrent** property.

Data Types: *FieldId* Integer

See also: **FieldLeft** Event.

FieldGoto Method

Description: Sets the current input position to the beginning of the specified marked text field and scrolls the text so that this position is at the top of the control's visible text.

Usage: TXTextControl.**FieldGoto** *FieldType, FieldIdOrName*

The method's parameters are:

Parameter	Description
FieldType	Specifies the type of the marked text field. See the Constants section of the **FieldType** property for valid values.
FieldIdOrName	Identifies the marked text field to which should be scrolled. It must be a valid field identifier. For fields of the type **txFieldLinkTarget** this parameter can also be the name of the field. For fields of the type

txFieldTopic this parameter can also be a valid topic number.

Return Value: If the field could be found the method returns True, otherwise it returns False.

Data Types:

FieldType	Integer
FieldIdOrName	Integer or String
Return value:	Boolean

FieldInsert Method

Description: Inserts a new marked text field at the current caret position.

Usage: TXTextControl.**FieldInsert** *FieldText*

Return Value: The method returns True if a field could be inserted, otherwise it returns False.

Remarks: Selected text can be converted to a marked text field by using an empty string as *FieldText*. Inserting a marked text field changes the value of the **FieldCurrent** property to the identifier of the newly created field.

Data Types:

FieldText	String
Return value:	Boolean

Example: See the description of the **FieldCurrent** property.

FieldLeft Event

Description: Occurs when the current input position indicated by the caret has been moved to a position that does not belong to the marked text field at the previous input position.

Syntax: **FieldLeft**(*FieldId*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field that has been left.

Remarks: This event that does not change the value of the **FieldCurrent** property.

Data Types: *FieldId* Integer

See also: **FieldEntered** Event.

FieldLinkClicked Event

Description: Occurs when a marked text field has been clicked on that represents the source of a hypertext link.

Syntax: **FieldLinkClicked**(*FieldId, FieldType, TypeData*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field that has been clicked on.
FieldType	Is the type of the field that has been clicked on. This event occurs only for fields of the types **txFieldExternalLink** and **txFieldInternalLink**.
TypeData	Specifies a character string that is the information to where the link points. This data has either been set with the **FieldTypeData** property or has been created through a text filter. For fields of the type **txFieldExternalLink** this can be any kind of address or file name. For fields of the type **txFieldInternalLink** this is the name of a marked text field of the type **txFieldLinkTarget**.

Data Types: *FieldId* Integer
FieldType Integer
TypeData String

See also: **FieldTypeData** Property

FieldNext Method

Description: This method returns the identifier of the marked text field that follows the specified field in the Text Control's current text. It can be used to find the next field in the text or to enumerate all fields. In a list of linked Text Controls the search is performed in all controls.

Usage: TXTextControl1.**FieldNext** *FieldId*, *FieldGroup*

The method's parameters are:

Parameter	Description
FieldId	Specifies a field identifier. If this parameter is zero the first field's identifier is returned.
FieldGroup	This parameter can be the sum of one or more constants used to seperate fields with certain attributes. Valid values are described in Remarks. If *FieldGroup* is zero, the method enumerates all fields.

Return Value: The method returns the identifier of the field that follows the specified field in the Text Control's text. It is zero if there is no following field.

Remarks: The settings for *Options* can include:

Setting	Description
0	Returns the identifiers of all fields.
&H1&	Returns only identifiers of fields which are both changeable and deleteable.
&H2&	Returns only identifiers of fields which are unchangeable.
&H4&	Returns only identifiers of fields which are undeleteable.
&H100&	Returns only identifiers of fields which have the type **txFieldLinkTarget**.
&H200&	Returns only identifiers of fields which have the type **txFieldExternalLink**.

&H400&	Returns only identifiers of fields which have the type **txFieldInternalLink**.
&H800&	Returns only identifiers of fields which have the type **txFieldPageNumber**.
&H1000&	Returns only identifiers of fields which have the type **txFieldHighlight**.
&H2000&	Returns only identifiers of fields which have the type **txFieldTopic**.

Data Types:

FieldId	Integer
FieldGroup	Integer
Return value:	Integer

See also: **FieldChangeable** Property, **FieldDeleteable** Property, **FieldType** Property

FieldPosX Property

Description: Returns the horizontal position of a marked text field. The field identifier must have previously been determined with the **FieldCurrent** property.

Usage: TXTextControl.**FieldPosX**

Remarks: The property value is the distance in horizontal direction between the left border of the marked text field and the left border of the Text Control. It is not affected by the scrollbar positions.

Limitations: Read only, run time only.

Data Type: Long.

See also: **FieldPosY** Property.

FieldPosY Property

Description: Returns the vertical position of a marked text field. The field identifier must have previously been determined with the **FieldCurrent** property.

Usage: TXTextControl.**FieldPosY**

Remarks: The property value is the distance in vertical direction between the upper left corner of the marked text field and the upper left corner of the text. It is not affected by the scrollbar positions.

Limitations: Read only, run time only.

Data Type: Long.

See also: **FieldPosX** Property.

FieldSetCursor Event

Description: Occurs when the cursor is moved over a marked text field.

Syntax: **FieldSetCursor**(*FieldId, MousePointer*)

The event procedure's parameters are:

Parameter	Description
FieldId	Is the identifier of the field where the cursor is moved over.
MousePointer	When this parameter is changed Text Control uses the specified cursor whilst moving over the marked text field. When this parameter is not changed, Text Control uses its standard cursor for marked text fields (Up Arrow). For possible values see the description of the *MousePointer* property.

Remarks: This event that does not change the value of the **FieldCurrent** property.

Data Types:

FieldId	Integer
MousePointer	Integer

See also: **FieldChanged** Event, **FieldClicked** Event, **FieldCreated** Event, **FieldDblClicked** Event, **FieldDeleted** Event.

FieldStart Property

Description: Specifies the start position of a marked text field. The field identifier must have previously been determined with the **FieldCurrent** property.

Usage: TXTextControl.**FieldStart**

Data Type: Long.

Limitations: Read only, run time only.

See also: **FieldEnd** Property.

FieldText Property

Description: Returns or sets the text which is contained within a marked text field. The field identifier must have previously been determined with the **FieldCurrent** property.

Usage: TXTextControl.**FieldText** [=*string*]

Data Type: String.

Limitations: Run time only.

FieldType Property

Description: This property sets or returns the type of a marked text field. The chapter "*Overviews - Marked Text Fields - Special Types of Marked Text Fields*" describes all the types and the data belonging to these types. Type-related data must be set with the **FieldTypeData** property

Usage: TXTextControl.**FieldType**(*FieldId*) [= *FieldType*]

The property's parameters are:

Parameter	Description
FieldId	Identifies the field that is to be manipulated.

Constants: The property setting can be one of the following constants:

Constant	Value	Description
txFieldExternalLink	1	Defines the source of a hypertext link to a location outside of the document.The **FieldTypeData** property must be used to define where the link points to.
txInternalLink	2	Defines the source of a hypertext link to a location in the same document.The **FieldTypeData** property must be used to define where the link points to. It must be the name of a marked text field that has the **txFieldLinkTarget** type.
txFieldPageNumber	3	This field displays the current page number. It can only be used in headers or footers.
txFieldLinkTarget	4	Defines a position in a document which is the target of a hypertext link. The **FieldTypeData** property must be used to define the name of this field.
txFieldHighlight	5	Defines a piece of text that can be highlighted. The **FieldTypeData** property must be used to define the color of the highlight.
txFieldTopic	6	Defines a position in a document that is the beginning of a topic. The **FieldTypeData** property must be used to define the number of the topic.
txFieldStandard	0	Resets a field of a special type to a standard marked text field.

Remarks: The types **txFieldLinkTarget, txFieldPageNumber** and **txFieldTopic** can only be set when the marked text field has no text.

Data Types: *FieldId* Integer
Property value: Integer.

Limitations: Run time only.

See also: **FieldInsert** Method, **FieldTypeData** Property, **FieldLinkClicked** Event.

FieldTypeData Property

Description: This property sets or returns the data that belongs to a marked text field of a special type. The chapter "*Overviews - Marked Text Fields - Special Types of Marked Text Fields*" informs about all the types and the data belonging to these types.

Usage: TXTextControl.**FieldTypeData**(*FieldId*) [= *TypeData*]

The property's parameters are:

Parameter	Description
FieldId	Identifies the field that is to be manipulated.

Remarks: The specified data can be a long value or a character string depending on the type of the field. A long value is used for fields of the types **txFieldHighlight** and **txFieldTopic**. For fields of the types **txFieldExternalLink**, **txFieldInternalLink** and **txFieldLinkTarget** the *TypeData* parameter must be a character string.

Data Types: *FieldId* Integer
Property value: Long or String.

Limitations: Run time only.

See also: **FieldInsert** Method, **FieldLinkClicked** Event.

Find Method

Description: Searches the text in a Text Control for a given string.

Usage: TXTextControl.**Find** *FindWhat[, Start[, Options]]*

The method's parameters are:

Parameter	Description
FindWhat	Specifies the string to search for.
Start	Optional. An integer character index that determines where to begin the search. The first character of text in the control has an index of 0. When this parameter is omitted or set to -1, the search begins at the current input position.
Options	Optional. Is the sum of one or more constants used to specify optional features, as described in Remarks.

Return Value: If the text searched for is found, the **Find** method highlights the specified text and returns the index (zero-based) of the first character highlighted. If the specified text is not found, the **Find** method returns -1.

Remarks: The settings for *Options* can include:

Setting	Description
1 - SearchUp	Determines the direction of searches through a document. If this flag is used, the search direction is up; if the flag is not used, the search direction is down.
4 - MatchCase	Determines if a match is based on the case of the specified string as well as the text of the string.
8 - NoHighLight	Determines if a match appears highlighted in the Text Control.
16 - NoMessageBox	Suppresses the built-in message boxes which inform the user that a match could not be found.

Data Types:

FindWhat:	String
Start:	Long
Options:	Long
Return Value:	Long

See also: **FindReplace** Method

FindReplace Method

Description: Displays a 'Find' or 'Replace' dialog box.

Usage: TXTextControl.**FindReplace** *TypeOfDialog*

Return Value: This method has no return value.

Remarks: The *TypeOfDialog* parameter can have one of the following values:

Value	Description
1	Displays a 'Find' dialog box.
2	Displays a 'Replace' dialog box.

Data Type: *TypeOfDialog* Integer

FontBold Property
FontItalic Property
FontStrikethru Property
FontUnderline Property

Description: Returns or sets font styles in the following formats: **Bold**, *Italic*, ~~Strikethru~~, and Underline. At design time these properties determine the styles of the complete text. At runtime these properties get or set the styles of the selected text when the **FormatSelection** property has been set to True. When the **FormatSelection** property has been set to False style settings are made for the complete text.

Usage: TXTextControl.**FontBold** [= *value*]
TXTextControl.**FontItalic** [= *value*]
TXTextControl.**FontStrikethru** [= *value*]
TXTextControl.**FontUnderline** [= *value*]

The properties' settings are:

Setting	Description
0	The characters are not formatted with the specified style.
1	The characters are formatted with the specified style.
2	Indicates that the selection contains characters that have a mix of the appropriate font styles. This value is only possible at runtime and when the **FormatSelection** property has been set to True.

Data Type: Integer.

See also: **FontName** Property, **FontSize** Property, **FontDialog** Method, **FormatSelection** Property, **FontUnderlineStyle** Property.

FontDialog Method

Description: Invokes the Text Control's built-in font dialog box and, after the user has closed the dialog box, specifies whether he has changed something.

Usage: TXTextControl.**FontDialog**

Return Value: The method returns True when the user has changed one or more attibutes. The method returns False when the formatting remains unchanged.

Remarks: The changes, made in the dialog box, apply to the currently selected text.

Data Types: Return value: Boolean.

FontName Property

Description: Returns or sets the font used to display text. At design time this property changes or gets the font name of the complete text. At runtime this property determines the font of the selected text when the **FormatSelection** property has been set to True. When the

FormatSelection property has been set to False the font of the complete text is set or returned.

Usage: TXTextControl.**FontName** [= *string*]

Remarks: The property returns an empty string if the selected text contains different fonts. This can happen only at runtime and when the **FormatSelection** property has been set to True.

Data Type: String.

See also: **FontDialog** Method, **FormatSelection** Property.

FontSize Property

Description: Returns or sets a value that specifies the size of the font used to display text. This value is in points. At design time this property changes or gets the font size of the complete text. At runtime this property determines the font size of the selected text when the **FormatSelection** property has been set to True. When the **FormatSelection** property has been set to False the font size of the complete text is set or returned.

Usage: TXTextControl.**FontSize** [= *value*]

Remarks: The property returns 0 if the selected text contains fonts with different sizes. This can happen only at runtime and when the **FormatSelection** property has been set to True.

Data Type: Integer.

See also: **FontDialog** Method, **FormatSelection** Property.

FontUnderlineStyle Property

Description: This property determines styles for the **FontUnderline** property.

Usage: TXTextControl.**FontUnderlineStyle** [= *value*]

The property's settings are:

Setting	Description
&H1&	Underline style is single underlined.

&H2&	Underline style is double underlined.
&H4&	Underline style is words only underlined. This value is possible only in combination with single or double underlined.
&H10&	The text contains single underlined parts.
&H20&	The text contains double underlined parts.
&H40&	The text contains parts that are underlined words only.

Remarks: The first group of values describes the underline styles. The second group are additional values that are only useful for the property return value. They inform about complex selections and are possible only when the **FormatSelection** property has been set to True.

Data Type: Integer.

Limitations: Run time only.

See also: **FontUnderline** Property, **FormatSelection** Property.

ForeColor Property

Description: Returns or sets the color used to display text in a Text Control. Text Control uses the Microsoft Windows operating enviroment red-green-blue (RGB) color scheme.
At design time this property changes or gets the text color of the complete text. At runtime this property determines the color of the selected text when the **FormatSelection** property has been set to True. When the **FormatSelection** property has been set to False the text color of the complete text is set or returned.

Usage: TXTextControl.**ForeColor** [= *value*]

The property's settings are:

Setting	Description
RGB colors	The valid range for a RGB color is 0 to &HFFFFFF. The high byte of a number in this range equals 0; the lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF).
System colors	Colors specified by the system color constants. If the high byte isn't 0, Text Control uses the system colors, as defined in the user's Control Panel settings.

Data Type: Long.

See also: **FormatSelection** Property, **BackColor** Property.

FormatSelection Property

Description: Specifies whether character and paragraph formatting properties apply to the whole text or to a particular selection only.

Usage: TXTextControl.**FormatSelection** [= *boolean*]

The property's settings are:

Setting	Description
True	The formatting properties only apply to selected text. This mode works only at run time, because at design time it is not possible to select text.
False	(Default) The formatting properties apply to the complete text.

Remarks: The properties which are affected are **Alignment**, **BaseLine**, **FontBold**, **FontItalic**, **FontName**, **FontSize**, **FontStrikethru**, **FontUnderline**,

FontUnderlineStyle, **LineSpacing**, **LineSpacingT**, **ForeColor**, **TextBkColor**.

Data Type: Boolean.

FrameDistance Property

Description: Returns or sets the distance between text and paragraph frame for the currently selected paragraph(s).

Usage: TXTextControl.**FrameDistance** [= *value*]

Remarks: The property returns -1 if the user selects two or more paragraphs which have different frame distance settings.

Data Type: Integer.

Limitations: Run time only.

FrameLineWidth Property

Description: Specifies the line widths of the currently selected paragraphs' frames.

Usage: TXTextControl.**FrameLineWidth** [= *value*]

Remarks: The property returns 0 if the user selects two or more paragraphs which have different line width settings.

Data Type: Integer.

Limitations: Run time only.

FrameStyle Property

Description: Returns or sets the style of the currently selected paragraphs' frames.

Usage: TXTextControl.**FrameStyle** [= *value*]

The property value can be a combination of the following values:

Setting	Description
BF_LEFTLINE (&H1)	Draws a left frame line.

BF_RIGHTLINE (&H2)	Draws a right frame line.
BF_TOPLINE (&H4)	Draws a top frame line.
BF_BOTTOMLINE (&H8)	Draws a bottom frame line.
BF_TABLINES (&H10)	Draws a vertical line at each tab position.
BF_SINGLE (&H20)	Draws a single line.
BF_DOUBLE (&H40)	Draws a doubled line.
BF_BOXCONNECT (&H80)	Draws a doubled line.
BF_NOLEFTLINE (&H100)	Resets an existing left line.
BF_NORIGHTLINE (&H200)	Resets an existing right line.
BF_NOTOPLINE (&H400)	Resets an existing top line.
BF_NOBOTTOMLINE (&H800)	Resets an existing bottom line.
BF_NOTABLINES (&H1000)	Resets existing tabulator lines.

Remarks: The property returns -1 if the user selects two or more paragraphs which have different frame style settings.

Data Type: Integer.

Limitations: Run time only.

HeaderFooter Property

Description: This property determines which kind of headers and/or footers the document contains. It can only be used when the **PageWidth** and **PageHeight** properties have non-zero values. Using this property, a header or footer is not automatically activated. The **HeaderFooterActivate** method or the built-in mouse interface can be used to activate a header or footer.

Usage: TXTextControl.**HeaderFooter** [= *HeadersFooters*]

Constants: The setting for *HeadersFooters* can be the sum of one or more of the following constants:

Constant	Value	Description
txHeader	1	Inserts a header.
txFirstHeader	2	Inserts a special header for the first page.
txFooter	4	Inserts a footer.
txFirstFooter	8	Inserts a special footer for the first page.

Data Type: Integer.

Limitations: Run time only.

See also: **HeaderFooterActivate** Method, **HeaderFooterStyle** Property

HeaderFooterActivate Method

Description: Activates or deactivates a header or a footer. During activation the current input focus is set to the header or footer area, so that the user can alter the text and/or the format. During deactivation the input focus is set back to the main text.

Usage: TXTextControl.**HeaderFooterActivate** *HeaderFooter*

The method's parameters are:

Parameter	Description
HeaderFooter	Specifies the header or footer to activate. Valid values are described in Constants. When this value is zero, a currently activated header or footer is deactivated.

Return Value: The method returns True, if a header or footer could be activated, otherwise it returns False.

Constants: The settings for *HeaderFooter* can be one of the following constants:

Constant	Value	Description
txHeader	1	Activates the header.
txFirstHeader	2	Activates the special header of the first page.

txFooter	4	Activates the footer.
txFirstFooter	8	Activates the special footer of the first page.

Data Types: *HeaderFooter* Integer
Return value: Boolean

HeaderFooterActivated Event

Description: Occurs when a header or footer has been activated.

Syntax: **HeaderFooterActivated**(*HeaderFooter*)

The event procedure's parameters are:

Parameter	Description
HeaderFooter	Specifies the header or footer that has been activated. Valid values are listed in Constants.

Constants: Valid values for *HeaderFooter* are:

Constant	Value	Description
txHeader	1	A header has been activated.
txFirstHeader	2	The special header for the first page has been activated.
txFooter	4	A footer has been activated.
txFirstFooter	8	The special footer for the first page has been activated.

Data Types: *HeaderFooter* Integer

See also: **HeaderFooterDeactivated** Event

HeaderFooterDeactivated Event

Description: Occurs when a header or footer has been deactivated.

Syntax: **HeaderFooterDeactivated**(*HeaderFooter*)

The event procedure's parameters are:

Parameter	Description
HeaderFooter	Specifies the header or footer that has been deactivated. Valid values are listed in Constants.

Constants: Valid values for *HeaderFooter* are:

Constant	Value	Description
txHeader	1	A header has been deactivated.
txFirstHeader	2	The special header for the first page has been deactivated.
txFooter	4	A footer has been deactivated.
txFirstFooter	8	The special footer for the first page has been deactivated.

Data Types: *HeaderFooter* Integer

See also: **HeaderFooterActivated** Event

HeaderFooterPosition Property

Description: This property specifies the position of a header or footer. For headers the position value is the distance between the top of the header and the top of the page. For footers the position value is the distance between the bottom of the footer and the bottom of the page. All values are in twips. The default value is 567 twips = 1 cm.

Usage: TXTextControl.**HeaderFooterPosition**(*HeaderFooter*) [= *position*]

Remarks: Valid settings for *HeaderFooter* are:

Constant	Value	Description
txHeader	1	Specifies the header.
txFirstHeader	2	Specifies the special header for the first page.
txFooter	4	Specifies the footer.
txFirstFooter	8	Specifies the special footer for the first page.

Data Type: Long.

Limitations: Run time only.

HeaderFooterSelect Method

Description: This method determines whether a certain Text Control property or method manipulates a header or a footer or the main text. This method does not activate a header or footer.

Usage: TXTextControl.**HeaderFooterSelect** *HeaderFooter*

The method's parameters are:

Parameter	Description
HeaderFooter	Specifies the part of the text that is to be selected. Valid values are described in Constants. When this value is zero, Text Control performs automatic selection, which means that a certain property or method manipulates the text part with the current input position.

Return Value: The method returns True, if a header or footer could be selected, otherwise it returns False.

Constants: The settings for *HeaderFooter* can be one of the following constants:

Constant	Value	Description
txHeader	1	Selects the header.
txFirstHeader	2	Selects the special header for the first page.
txFooter	4	Selects the footer.
txFirstFooter	8	Selects the special footer for the first page.
txMainText	10	Selects the main text.

Data Types:

HeaderFooter	Integer
Return value:	Boolean

Example: This Basic example selects the header, alters the text of the header and returns to the automatic mode:

```
TXTextControl1.HeaderFooterSelect txHeader
TXTextControl1.Text = "This is the header's text"
TXTextControl1.HeaderFooterSelect 0
```

HeaderFooterStyle Property

Description: This property determines how headers and footers can be activated and how activated headers and footers appear on the screen.

Usage: TXTextControl.**HeaderFooterStyle** [= *style*]

Remarks: Valid settings are the sum of one or more constants specified in the folowing list:

Constant	Value	Description
txMouseClick	1	Headers and footers can be activated through single mouse clicks.
txNoDblClk	2	Headers and footers cannot be activated through mouse double-clicks.
txSolidFrame	4	An activated header or footer has a solid border to indicate its size.
txUnframed	8	An activated header or footer has no border.

The default style setting is a dotted border for an activated header or footer and a mouse interface that activates a header or footer with double-clicks.

Data Type: Integer.

Limitations: Run time only.

HExpand Event

Description: Occurs when the Text Control has changed its window size horizontally. This event can only occur when the **AutoExpand** property is set to True.

Syntax: **HExpand**( )

See also: **AutoExpand** Property, **VExpand** Event.

HideSelection Property

Description: Specifies whether a text selection is hidden when the Text Control looses the input focus.

Usage: TXTextControl.**HideSelection** [= *boolean*]

The property's settings are:

Setting	Description
True	(Default) The selection is hidden when the Text Control looses the input focus and shown when the Text Control gets the input focus.
False	The selection stays visible, independent of the current input focus.

Data Type: Boolean.

HScroll Event

Description: Occurs when the horizontal scroll position has been changed.

Syntax: **HScroll**( )

See also: **VScroll** Event.

hWnd Property

Description: Returns a handle to a Text Control.

Usage: TXTextControl.**hWnd** [= *handle*]

Remarks: The **hWnd** property is used with Windows API calls. Many Windows operating enviroment functions require the **hWnd** of the active window as an argument.

Data Type: Handle.

Limitations: Read only, run time only.

ImageDisplayMode Property

Description: Provides several modes how images are displayed or refreshed.

Usage: TXTextControl.**ImageDisplayMode** [= *value*]

The property's settings are:

Setting	Description
0	(Default) Standard mode.
&H1&	Displays an image as a gray rectangle.
&H2&	Container mode. This mode ensures proper refreshing when the image is used as a background image for transparent controls. This mode is read only and can only be set with the **ObjectInsertFixed** and **ObjectInsertAsChar** methods.

Remarks: The property settings can be combined by adding the desired constant values.

Data Type: Integer.

Limitations: Run time only.

See also: **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method.

ImageFilename Property

Description: Determines the image filename of an embedded object. This property is only available when the object has been inserted as a Image Control

(See the **ObjectInsertAsChar** method for more information). The object identifier must have previously been determined with the **ObjectCurrent** property.

Usage: TXTextControl.**ImageFilename** [= *string*]

Data Type: String.

Limitations: Run time only.

See also: **ObjectCurrent** Property, **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method.

ImageFilters Property

Description: This property returns a string which specifies the available image filters. This string can be used to initialize the **Filter** property of a Common Dialog control.

Usage: TXTextControl.**ImageFilters**

Remarks: The filter names are read from the IC.INI or IC32.INI files. In the standard version, only filters for TIFF, Bitmap and Windows Metafiles are supplied, but you can add any Aldus compatible image filter library, for instance Visual Tools VTImageStream.

Data Type: String.

Limitations: Read only, run time only.

See also: **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method.

Example: This Basic example initializes an **OpenFile** dialog box with the names of the available image filters and then loads a selected image:

```
dlgFile.DialogTitle = "Insert Image"
dlgFile.Filename = ""
dlgFile.Filter = TXTextControl1.ImageFilters
dlgFile.FilterIndex = 1
dlgFile.Flags = cdlOFNPathMustExist Or cdlOFNFileMustExist _
Or cdlOFNHideReadOnly
dlgFile.CancelError = True
dlgFile.ShowOpen
```

```
TXTextControl1.ObjectInsertAsChar 0, dlgFile.Filename, _
   1, 100, 100, 0, 0
```

ImageSaveMode Property

Description: When saving a Text Control file using the **Save** or **RTFExport** methods, this property determines whether the image data or the image file name is stored.

Usage: TXTextControl.**ImageSaveMode** [= *value*]

The property's settings are:

Setting	Description
0	Saves the image filename.
1	(Default) Saves the image data.

Data Type: Integer.

Limitations: Run time only.

See also: **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method, **Save** Method.

IndentB Property
IndentFL Property
IndentL Property
IndentR Property
IndentT Property

Description: Returns or sets the left, top, right, bottom, and first line indents (in twips) for a paragraph or a selected range of paragraphs.

Usage: TXTextControl.**IndentB** [= *value*]
TXTextControl.**IndentFL** [= *value*]
TXTextControl.**IndentL** [= *value*]

TXTextControl.**IndentR** [= *value*]
TXTextControl.**IndentT** [= *value*]

Remarks: If a number of paragraphs have been selected which have different settings for one of the indents, the appropriate property returns &H8000. The first line indent can be negative.

Data Type: Integer.

InputPosFromPoint Method

Description: Returns the text input position belonging to a certain geometric position. The text input position is relative to the beginning of the text and the geometric position is a position in the visible part of the text.

Usage: TXTextControl.**InputPosFromPoint** *X, Y*

The method's parameters are:

Parameter	Description
X, Y	Specify the coordinates of the point. These values must be in twips.

Return Value: The method returns the text input position beginning with zero for the position in front of the first character. The method returns -1, if a text position could not be found.

Data Types:

X, Y	Long
Return value:	Long

InsertionMode Property

Description: Specifies whether text is inserted or overwrites existing text.

Usage: TXTextControl.**InsertionMode** [= *boolean*]

The property's settings are:

Setting	Description
True	(Default) New text is inserted.

False	New text overwrites existing text.

Data Type: Boolean.

KeyDown Event
KeyUp Event

Description: Occurs when the user presses (KeyDown) or releases (KeyUp) a key while the Text Control has the input focus. To interpret ANSI characters, use the **KeyPress** event.

Syntax: **KeyDown**(*KeyCode, Shift*)
KeyUp(*KeyCode, Shift*)

The event procedures' parameters are:

Parameter	Description
KeyCode	Is the virtual-key code of the pressed or released key. When this value is changed, Text Control handles the changed key.
Shift	Informs about the state of the SHIFT, CTRL and ALT keys at the time of the event. It is the sum of one or more of the values described in Remarks.

Remarks: The settings for *Shift* can include:

Value	Description
1	The SHIFT key was pressed at the time of the event.
2	The CTRL key was pressed at the time of the event.
4	The ALT key was pressed at the time of the event.

Data Types:

KeyCode	Integer
Shift	Integer

See also: **KeyPress** Event

KeyPress Event

Description: Occurs when the user presses and releases an ANSI key while the Text Control has the input focus.

Syntax: **KeyPress**(*KeyAscii*)

The event procedure's parameters are:

Parameter	Description
KeyAscii	Is a standard numeric ANSI keycode. Changing it sends a different character to the Text Control.

Data Types: *KeyAscii* Integer

See also: **KeyDown** Event, **KeyUp** Event

KeyStateChange Event

Description: Occurs when the character insertion mode or when the state of the NUMLOCK or CAPSLOCK key has been changed.

Syntax: **KeyStateChange**( )

Language Property

Description: Determines the language in which Text Control displays dialog boxes and error messages. Text Control has several built-in languages, additional languages can be added with the **ResourceFile** property. The default language is the Windows system language. See "*Overviews - Resources*" for more information.

Usage: TXTextControl.**Language** [= *value]*

The property's settings are:

Setting	Description
33	French
34	Spanish

39	Italian
41	German (Switzerland)
43	German (Austria)
49	German
81	Japanese (32 bit only)
else	English

Data Type: Integer.

LineSpacing Property

Description: Specifies the line spacing for the currently selected paragraphs as a percentage of the font size.

Usage: TXTextControl.**LineSpacing** [= *value*]

Data Type: Integer.

See also: **FormatSelection** Property.

LineSpacingT Property

Description: Specifies the line spacing for the currently selected paragraphs in twips.

Usage: TXTextControl.**LineSpacingT** [= *value*]

Data Type: Integer.

See also: **FormatSelection** Property.

Load Method

Description: Loads data from a file and inserts it into a Text Control. All Unicode formats (6, 7 and 8) and the Microsoft Word format can only be used with the 32 bit version of Text Control.

Usage: TXTextControl.**Load** *FileName[, Offset[, Format[, CurSelection]]]*

The method's parameters are:

Parameter	Description
FileName	Specifies the file to load from.
Offset	Optional. Specifies the file position from where the data is read. When not specified or set to zero the data is read from the beginning.
Format	Optional. Specifies a format identifier or the name of a user-developed filter. When not specified Text Control assumes TX format (id. 3). The following format identifiers are possible: 1 - ANSI text: Text only in ANSI format (Windows compatible). 2 - TX text: Text only in ANSI format (Text Control compatible). 3 - TX: Internal Text Control format. 4 - HTML: HTML format (Hypertext Markup Language). 5 - RTF: RTF format (Rich Text Format). 6 - Unicode text: Text only in Unicode format (Windows compatible). 7 - TX text: Text only in Unicode format (Text Control compatible). 8 - TX: Internal Text Control format. Text is stored in Unicode. 9 - WORD: Microsoft Word format. *FilterFileName*: Can be used with a user-developed filter.

CurSelection	Optional. When set to true the loaded data replaces the current selection or is inserted at the current input position. The new input position is behind the inserted data. When omitted or set to false the loaded data replaces the complete control contents independent of the current selection. The new input position is at the beginning of the data.

Return Value: The method returns the position in the file after the data has been loaded.

Data Type:

FileName:	String
Offset:	Long
Format:	Integer or String
CurSelection:	Boolean
Return value:	Long

See also: **Save** Method.

Example: This Basic example opens a file and loads its contents into TXTextControl1:

```
Private Sub mnuFile_Load_Click()
   On Error Resume Next
   ' Create an "Open File" dialog box
   CommonDialog1.Filter = "TX Demo (*.tx)|*.tx"
   CommonDialog1.DialogTitle = "Open"
   CommonDialog1.Flags = cdlOFNFileMustExist Or _
      cdlOFNHideReadOnly
   CommonDialog1.CancelError = True
   CommonDialog1.ShowOpen
   If Err Then Exit Sub
   ' Pass the filename to the text control
   TXTextControl1.Load CommonDialog1.filename
End Sub
```

LoadFromMemory Method

Description: Loads data from a byte array and inserts it into a Text Control. This method works in the same way as the **Load** method.

Usage: TXTextControl.**LoadFromMemory** *DataArray[, Format[, CurSelection]]*

The method's parameters are:

Parameter	Description
DataArray	Specifies the byte array to load from.
Format	Optional. See the *Format* parameter of the **Load** method for a description.
CurSelection	Optional. See the *CurSelection* parameter of the **Load** method for a description.

Return Value: The method returns True if the data could be loaded, otherwise it returns False.

Data Types:

DataArray:	One-dimensional Byte Array
Format:	Integer or String
CurSelection:	Boolean
Return value:	Boolean

See also: **Load** Method, **SaveToMemory** Method

LoadSaveAttribute Property

Description: This property enables an application to specify several attributes that can be used in combination with the **Load** and **Save** methods for the following situations:

1. Documents can contain elements, for example a document title, that cannot be converted directly to text properties but may be useful for certain applications. Such an attribute can be set before a document is saved, in order to store it as part of the document, or it can be provided and used after a document has been loaded.

2. In some documents generic information about text properties is left out, for example with HTML documents which define font heights as a percentage of a base height without specifying that base height. Such an attribute can be set before a document is loaded, in order to inform the filter how to calculate such relative values.

Usage: TXTextControl.**LoadSaveAttribute**(*Attribute*) [= *value*]

The property's parameters are:

Setting	Description
Attribute	Determines the attribute that is to be changed or returned. Possible values are listed in Constants.

Constants: Valid values for *Attribute* are:

Constant	Description
txDocWidth	Loads or saves a document width in twentieths of a point. When this property is set before a document is loaded, it is used to calculate width values contained in the document which are relative to the document's width. After a document has been loaded, this property returns the width, contained in the document, or -1 if the document does not contain a width. When a document is saved and this property has not been set, Text Control saves the value of the **PageWidth** property instead, except this property has been set to -1 previously.
txDocHeight	Loads or saves a document height in twentieths of a point. It is used in the same manner as described for **the txDocWidth** attribute.
txDocLeftMargin	Loads or saves a left document margin in twentieths of a point. After a document has been loaded, this property returns the margin contained in the document, or -1 if the

	document does not contain the margin. When a document is saved and this property has not been set, Text Control saves the value of the **PageMarginL** property instead, except this property has been set to -1 previously.
txDocTopMargin	Loads or saves a top document margin in twentieths of a point. It is used in the same manner as described for the **txDocLeftMargin** attribute.
txDocRightMargin	Loads or saves a right document margin in twentieths of a point. It is used in the same manner as described for the **txDocLeftMargin** attribute.
txDocBottomMargin	Loads or saves a bottom document margin in twentieths of a point. It is used in the same manner as described for the **txDocLeftMargin** attribute.
txDocTitle	Loads or saves a document title. After a document has been loaded, this property returns the document title contained in the document, or an empty string if the document does not contain a title.
txDocBkColor	Loads or saves a document background color as a RGB value. After a document has been loaded, this property returns the background color contained in the document, or -1, if the document does not contain a background color.
txAbsPath	Specifies a character string that is used to search for resources like images or destinations of hypertext links. When a document is loaded, this path is used to locate a resource. It is only used for resources which are specified through an absolute location. In

	this case the absolute resource location is completely replaced through the path specified through this property. When a document is saved, this attribute is not used.
txBasePath	Specifies a character string that is used to search for resources given through a relative location. When a document is loaded, this path is added to the relative location of a resource. When a document is saved the string can only be a file path. All files in the document are saved relative to this path.
txBaseFontSize	HTML only. Specifies a base font size in points and is used to convert percentage font sizes to absolute font sizes. When not set, a value of 10 points is used. This attribute is only used when a document is loaded.
txPropFontName	HTML only. Defines a proportional font name when not specified in the document. When not set, Text Control uses a default font. This attribute is only used when a document is loaded.
txMonoFontName	HTML only. Defines a mono-spaced font name when not specified in the document. When not set, Text Control uses a default mono-spaced font. This attribute is only used when a document is loaded.
txTextColor	HTML only. Defines a text color. This attribute is only used when a document is loaded. When the **txOverwriteTextColor** attribute is set to True, this color is used for text coloring. Otherwise when not set to overwrite, this color is only used when no text color is specified in the document.
txOverwriteTextColor	HTML only. Sets the **txTextColor** attribute to overwrite or not to overwrite.

txTextBkColor HTML only. Defines a text background color. This attribute is only used when a document is loaded. When the **txOverwriteTextBkColor** attribute is set to True, this color is used for text background coloring. Otherwise, when not set to overwrite, this color is only used when no text background color is specified in the document.

txOverwriteTextBkColor HTML only. Sets the **txTextBkColor** attribute to overwrite or not to overwrite.

txLoadImages HTML only. Specifies whether or not images are loaded. When not set, images are replaced by its alternate text or a special link text. For all other formats images are always loaded.

txEnableLinks HTML, RTF and Word only. Converts source and target fields of hypertext links to appropriate marked text fields.

txEnableHighlights RTF only. Converts all '\cb*N*' keywords into marked text fields of the type **txFieldHighlight**.

txEnableTopics RTF only. Converts all '\sect' keywords into marked text fields of the type **txFieldTopic**.

txLinkColor HTML only. Defines a text color for pieces of text which function as hypertext links. This attribute is only used when a document is loaded. When the **txOverwriteLinkColor** attribute is set to True, this color is used to color hypertext links. Otherwise when not set to overwrite, this color is only used when no color for links is specified in the document.

txOverwriteLinkColor HTML only. Sets the **txLinkColor** attribute to overwrite or not to overwrite.

txUnderlineLinks HTML only. Specifies whether or not hypertext links are underlined. This attribute

is only used when a document is loaded. When set to False, hypertext links are only underlined when specified in the document.

Data Types: The following lists the data type for each attribute including its numeric value:

Constant	Value	DataType
txDocWidth	0	Long
txDocHeight	1	Long
txDocLeftMargin	2	Long
txDocTopMargin	3	Long
txDocRightMargin	4	Long
txDocBottomMargin	5	Long
txDocTitle	6	String
txDocBkColor	7	Long
txAbsPath	28	String
txBasePath	29	String
txBaseFontSize	30	Integer
txPropFontName	31	String
txMonoFontName	32	String
txTextColor	33	Long
txOverwriteTextColor	34	Boolean
txTextBkColor	35	Long
txOverwriteTextBkColor	36	Boolean
txLoadImages	37	Boolean
txEnableLinks	38	Boolean
txEnableHighlights	39	Boolean
txEnableTopics	40	Boolean
txLinkColor	50	Long
txOverwriteLinkColor	51	Boolean
txUnderlineLinks	52	Boolean

Limitations: Run time only

See also: **Load** Method, **Save** Method

MouseDown Event

MouseMove Event
MouseUp Event

Description: Occurs when the user presses (MouseDown) or releases (MouseUp) a mouse button or when the user moves the mouse (MouseMove).

Syntax: **MouseDown**(*Button, Shift, X, Y*)
MouseMove(*Button, Shift, X, Y*)
MouseUp(*Button, Shift, X, Y*)

The event procedures' parameters are:

Parameter	Description
Button	Informs about the state of the mouse buttons at the time of the event. Possible settings are described in Remarks.
Shift	Informs about the state of the SHIFT, CTRL and ALT keys at the time of the event. Possible values are the same as for the **KeyDown** and **KeyUp** events.
X,Y	Specifies the current location of the mouse pointer. The coordinates are relative to the upper-left corner of the Text Control's window.

Remarks: The following are valid values for *the Button* parameter. For the **MouseDown** and **MouseUp** events it can only be one of the values. For the **MouseMove** event it can be the sum of more than one value.

Value	Description
1	The left button is pressed.
2	The right button is pressed.
4	The middle button is pressed.

Data Types:

Button	Integer
Shift	Integer

X	Long
Y	Long

See also: **KeyDown** Event, **KeyUp** Event.

MousePointer Property

Description: Returns or sets a value indicating the type of mouse pointer displayed when the mouse is over a particular part of a Text Control at run time.

Usage: TXTextControl.**MousePointer** [= *value]*

The property's settings are:

Setting	Description
0	(Default) The mouse pointer is an I-Beam in edit mode and an arrow in read-only mode. See the **EditMode** property for more information.
1	Arrow.
2	Cross.
3	I-Beam.
4	Icon (small square within a square).
5	Size (four-pointed arrow).
6	Size NE SW (double arrow pointing northeast and southwest).
7	Size N S (double arrow pointing north and south).
8	Size NW SE (double arrow pointing northwest and southeast).
9	Size W E (double arrow pointing west and east).
10	Up Arrow.
11	Hourglass.
12	Hand.

Data Type: Integer.

Move Event

Description: Occurs when a Text Control has been moved with the mouse while depressing the ALT key.

Syntax: **Move**( )

See also: **Size** Event, **SizeMode** Property.

NextWindow Property

Description: Returns or sets the next window in a chain of linked windows.

Usage: TXTextControl.**NextWindow** [= *handle*]

Data Type: Handle.

Limitations: Run time only.

Example: The following Basic code line links 2 Text Controls so that text flows from TextControl1 to TextControl2:

```
TextControl1.NextWindow = TextControl2.hWnd
```

ObjectClicked Event

Description: Occurs when an object has been clicked on.

Syntax: **ObjectClicked**(*ObjectId*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that has been clicked.

Remarks: The value of the **ObjectCurrent** property is updated with the identifier given through the *ObjectId* parameter.

Data Types: *ObjectId* Integer

See also: **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method, **ObjectDblClicked** Event.

ObjectCreated Event

Description: This event specifies that a new object has been created. This can happen when a document that contains objects, is loaded or when an object is pasted from the clipboard. This event does not occur after inserting a new object with the **ObjectInsertFixed** or **ObjectInsertAsChar** methods.

Syntax: **ObjectCreated**(*ObjectId*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that has been created.

Remarks: The value of the **ObjectCurrent** property is updated with the identifier given through the *ObjectId* parameter.

Data Types: *ObjectId* Integer

See also: **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method

ObjectCurrent Property

Description: Returns or sets the current object for the **Objectxxx** and **Imagexxx** properties, methods, and events, except **ObjectInsertAsChar** and **ObjectInsertFixed**. The value is automatically updated when an object is inserted or when you click on an object.

Usage: TXTextControl.**ObjectCurrent** [= *ObjectId*]

Limitations: Run time only.

Data Type: Integer.

ObjectDblClicked Event

Description: Occurs when an object has been double-clicked on.

Syntax: **ObjectDblClicked**(*ObjectId*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that has been double-clicked.

Remarks: The value of the **ObjectCurrent** property is updated with the identifier given through the *ObjectId* parameter.

Data Types: *ObjectId* Integer

See also: **ObjectClicked** Event.

ObjectDelete Method

Description: This method deletes the object with the specified object identifier.

Usage: TXTextControl.**ObjectDelete** *ObjectId*

Return Value: The method returns True if the specified object could be deleted, otherwise it returns False.

Data Types: *ObjectId* Integer
Return value: Boolean

See also: **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method, **ObjectDeleted** Event.

ObjectDeleted Event

Description: Occurs when an object has been deleted.

Syntax: **ObjectDeleted**(*ObjectId*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that has been deleted.

Remarks: The value of the **ObjectCurrent** property is set to zero.

Data Types: *ObjectId* Integer

See also: **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method, **ObjectDelete** Method.

ObjectDistance Property

Description: Specifies the distance (in twips) between an object and the text that flows around it. This property can only be used with objects that have been inserted using the **ObjectInsertFixed** method. Otherwise an **Error** event is generated.

Usage: TXTextControl.**ObjectDistance**(*index*) [= *value*]

The property's parameters are:

Parameter	Description
index	Specifies one of the four possible distances: left (1), top (2), right (3), bottom (4).

Data Type: Integer.

Limitations: Run time only.

See also: **ObjectInsertAsChar** Method, **ObjectInsertFixed** Method.

ObjectGetData Event

Description: Occurs when a document which contains objects, is saved. This event is sent only for objects that have been inserted via its **hWnd** property. In response to this event, the application can store the object's data by copying it into the *ObjectData* parameter.

Syntax: **ObjectGetData**(*ObjectId, ObjectData*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that is to be saved.
ObjectData	The object's private data can be copied to this parameter.

Remarks: It is recommended to store binary data as a byte array and not as a string. If you want compatibility between the 16 bit and 32 bit version you should store strings always as ANSI strings.

Data Types:

ObjectId	Integer
ObjectData	Variant

See also: **ObjectSetData** Event, **ObjectGethWnd** Event

ObjectGethWnd Event

Description: Occurs when a document which contains objects, is loaded. This event is sent only for objects that have been inserted via its **hWnd** property. The application must create the object and copy the object's **hWnd** property to the *hWnd* parameter.

Syntax: **ObjectGethWnd**(*ObjectId*, *KindOfObject*, *hWnd*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that is to be created.
KindOfObject	Is the value that has been specified as *KindOfObject* parameter for the **ObjectInsertAsChar** or **ObjectInsertFixed** method.
hWnd	The **hWnd** property of the new created object must be copied to this variable.

Data Types:

ObjectId	Integer
KindOfObject	Integer
hWnd	Handle

See also: **ObjectGetData** Event, **ObjectSetData** Event.

ObjectGetZoom Event

Description: Occurs when an object's zoom factor is requested. This event is sent only for objects that have been inserted via its **hWnd** property.

Syntax: **ObjectGetZoom**(*ObjectId, ZoomFactor*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object, the zoom factor of which is requested.
ZoomFactor	The zoom factor of the object must be copied to this variable.

Data Types:

ObjectId	Integer
ZoomFactor	Integer

See also: **ObjectSetZoom** Event

ObjectInsertAsChar Method

Description: This method embeds a new object or image which is then handled like a single character in the text.

Usage: TXTextControl.**ObjectInsertAsChar** *hWnd, FileName, TextPos, ScaleX, ScaleY, ImageDisplayMode, ImageSaveMode [,KindOfObject]*

The method's parameters are:

Parameter	Description
hWnd	Specifies an externally created window that represents the object to be inserted. It can also be one of the following identifiers:

	0 - Image Control The Text Control creates an Image Control window and handles this window internally. In this case the *FileName* parameter must specify a file containing an image. 1 - OLE object Inserts an OLE object. The type of object can be selected with the system embedded *OLE Insert* dialog box (32 bit only). 2 - OLE object (programmatic identifier) Creates a newly created OLE object. In this case the *FileName* parameter must specify a string which is the programmatic identifier of the OLE object to insert.The programmatic identifier is stored under the *ProgID* key in the registration database. For example the programmatic identifier of a Text Control 5.0 is TX.TextControl.110 (32 bit only). 3 - OLE object (embedded) Inserts a newly created embedded OLE object from a file. In this case the *FileName* parameter must specify a valid filename (32 bit only). 4 - OLE object (linked) Inserts a newly created linked OLE object from a file. In this case the *FileName* parameter must specify a valid filename (32 bit only).
FileName	Specifies the full DOS path name of a file that contains an image. This parameter can be zero if *hWnd* specifies an externally created window.
TextPos	This parameter specifies the text position where the object is to be inserted. If *TextPos* is

	-1 the object is inserted at the current input position.
ScaleX	Specifies a horizontal scaling factor as a percentage. It must be a value between 10 and 250.
ScaleY	Specifies a vertical scaling factor as a percentage. It must be a value between 10 and 250.
ImageDisplayMode	see **ImageDisplayMode** Property.
ImageSaveMode	see **ImageSaveMode** Property.
KindOfObject	Optional. Specifies an identifier that is to be used with externally created windows. When a document with external windows is loaded an **ObjectGethWnd** event occurs for each window to give an application the opportunity of recreating these windows. This parameter can be used to handle groups of different types of windows.

Return Value: The method returns the object's identifier when an object could be inserted. Otherwise it returns zero. The object's identifier can also be obtained with the **ObjectCurrent** property.

Data Types:

hWnd:	Handle
FileName:	String
TextPos:	Long
ScaleX:	Integer
ScaleY:	Integer
ImageDisplayMode:	Integer
ImageSaveMode:	Integer
KindOfObject:	Integer
Return value:	Integer

See also: **ObjectInsertFixed** Method.

ObjectInsertFixed Method

Description: This method embeds a new object or image at a fixed position. The text flows around the object.

Usage: TXTextControl.**ObjectInsertFixed** *hWnd, FileName, PosX, PosY, ScaleX, ScaleY, ImageDisplayMode, ImageSaveMode, SizeMode, TextFlow, DistanceL, DistanceT, DistanceR, DistanceB[,KindOfObject]*

The method's parameters are:

Parameter	Description
hWnd	Specifies an externally created window that represents the object to be inserted. It can also be an identifier to insert Image Controls or OLE objects. See the *hWnd* parameter description of the **ObjectInsertAsChar** method for more information.
FileName	Specifies the full DOS path name of a file that contains an image. This parameter can be zero if *hWnd* specifies an externally created window.
PosX	Specifies the object's horizontal position in twentieths of a point.
PosY	Specifies the object's vertical position in twentieths of a point.
ScaleX	Specifies a horizontal scaling factor as a percentage. It must be a value between 10 and 250.
ScaleY	Specifies a vertical scaling factor as a percentage. It must be a value between 10 and 250.
ImageDisplayMode	see **ImageDisplayMode** Property.
ImageSaveMode	see **ImageSaveMode** Property.
SizeMode	see **ObjectSizeMode** Property.
TextFlow	see **ObjectTextFlow** Property.

DistanceL, *DistanceT,* *DistanceR,* *DistanceB*	see **ObjectDistance** Property.
KindOfObject	Optional. Specifies an identifier that is to be used with externally created windows. When a document with external windows is loaded an **ObjectGethWnd** event occurs for each window to give an application the opportunity of recreating the external windows. This parameter can be used to handle groups of different types of windows.

Return Value: The method returns the object's identifier when an object could be inserted. Otherwise it returns zero. The object's identifier can also be obtained with the **ObjectCurrent** property.

Data Types:

hWnd:	Handle
FileName:	String
PosX:	Long
PosY:	Long
ScaleX:	Integer
ScaleY:	Integer
ImageDisplayMode:	Integer
ImageSaveMode:	Integer
SizeMode:	Integer
TextFlow:	Integer
DistanceL:	Integer
DistanceT:	Integer
DistanceR:	Integer
DistanceB:	Integer
KindOfObject:	Integer
Return value:	Integer

See also: **ObjectInsertAsChar** Method.

ObjectItem Property

Description: Returns a reference to the object currently set with the **ObjectCurrent** property. This property is only available for inserted OLE objects.

Usage: TXTextControl.**ObjectItem**

Data Type: Object.

Limitations: Read only, run time only.

Example: The following Basic example inserts a Text Control into another Text Control at the current input position and sets the font bold attribute for the inserted Text Control. The **ObjectInsertAsChar** method implicitely sets the **ObjectCurrent** property to the just inserted object.

```
TXTextControl1.ObjectInsertAsChar 2, "TX.TextControl.110", -
1, 100, 100, 0, 0

TXTextControl1.ObjectItem.FontBold = True
```

ObjectMoved Event

Description: Occurs when an inserted object has been moved with the mouse while depressing the ALT key.

Syntax: **ObjectMoved**(*ObjectId*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that has been moved.

Data Types: *ObjectId* Integer

See also: **ObjectSized** Event.

ObjectNext Method

Description: This method returns the identifier of the object that follows the specified object in the Text Control's internal list of objects.

Usage: TXTextControl1.**ObjectNext** *ObjectId, ObjectGroup*

The method's parameters are:

Parameter	Description
ObjectId	Specifies a unique identifier returned by the **ObjectInsertAsChar** or **ObjectInsertFixed** method. If this parameter is zero the first object's identifier is returned.
ObjectGroup	This parameter specifies which kinds of objects are to be returned. It can be a sum of the values, described in Remarks. If this parameter is zero, the identifiers of all objects are returned.

Return Value: The method returns the next object's identifier. It returns zero when there is no following object.

Remarks: The settings for *ObjectGroup* can include:

Value	Description
1	Returns only identifiers of fixed positioned objects.
2	Returns only identifiers of objects that act as single characters.
4	Returns only identifiers of objects which are internally created by the Text Control using the Image-Control module.
8	Returns only identifiers of objects which are externally created by the application.

Data Types:

ObjectId:	Integer
ObjectGroup:	Integer
Return value:	Integer

ObjectPrint Event

Description: Occurs when a document which contains objects, is printed. This event is sent only for objects that have been inserted via its **hWnd** property.

Syntax: **ObjectPrint**(*ObjectId*, *Device*, *Left*, *Top*, *Right*, *Bottom*, *Processed*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that is to be printed.
Device	Is the printer device context.
Left, Top, Right, Bottom	Is the object's bounding rectangle. This rectangle is given in device pixels with an origin at the upper left corner of the object.
Processed	When the object has been printed, this parameter should be set to True.

Data Types:

ObjectId	Integer
Device	Long
Left, Top, Right, Bottom	Long
Processed	Boolean

See also: **ObjectGetData** Event, **ObjectSetData** Event, **ObjectGethWnd** Event

ObjectScaleX Property
ObjectScaleY Property

Description: Specifies the object's scaling factor as a percentage in the range of 10 to 400%. The object must have previously been selected with the **ObjectCurrent** property.

Usage: TXTextControl.**ObjectScaleX** [= *value*]
TXTextControl.**ObjectScaleY** [= *value*]

Data Type: Integer.

Limitations: Run time only.

See also: **ObjectDistance** Property, **ObjectInsertFixed** Method.

ObjectScrollOut Event

Description: Occurs when an object is scrolled out of the visible area.

Syntax: **ObjectScrollOut**(*ObjectId*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that has been scrolled.

Data Types: *ObjectId* Integer

See also: **ObjectMoved** Event, **ObjectSized** Event

ObjectSetData Event

Description: Occurs when a document which contains objects, is loaded. This event is sent only for objects that have been inserted via its **hWnd** property.

Syntax: **ObjectSetData**(*ObjectId*, *ObjectData*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that is loaded.
ObjectData	The data of the object in the format saved with the **ObjectGetData** event.

Data Types: *ObjectId* Integer
ObjectData Variant

See also: **ObjectGetData** Event, **ObjectGethWnd** Event

ObjectSetZoom Event

Description: Occurs when an object's zoom factor is to be changed. This event is sent only for objects that have been inserted via its **hWnd** property.

Syntax: **ObjectSetZoom**(*ObjectId*, *ZoomFactor*, *Processed*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object, the zoom factor of which is to be changed.
ZoomFactor	Is the object's new zoom factor.
Processed	If the event is being processed this parameter should be set to True.

Data Types:

ObjectId	Integer
ZoomFactor	Integer
Processed	Boolean

See also: **ObjectGetZoom** Event.

ObjectSized Event

Description: Occurs when an embedded object has been resized with the mouse while depressing the ALT key.

Syntax: **ObjectSized**(*ObjectId*)

The event procedure's parameters are:

Parameter	Description
ObjectId	Is the identifier of the object that has been sized.

Data Types:

ObjectId	Integer

See also: **ObjectMoved** Event.

ObjectSizeMode Property

Description: Specifies whether an inserted object can be moved or resized at run time. If the Moveable option is selected, the control can be moved on the background by depressing the ALT key and then dragging the control with the mouse. If the Sizeable option is selected and the ALT key is depressed, the borders of the control can be dragged.

Usage: TXTextControl.**ObjectSizeMode** [= *value*]

The property's settings are:

Setting	Description
0 - Fixed	(Default) The object cannot be moved or sized.
1 - Moveable	The object can be moved.
2 - Sizeable	The object can be sized.
3 - Move and Sizeable	The object can be moved and sized.

Data Type: Integer.

Limitations: Run time only.

See also: **ObjectMoved** Event, **ObjectSized** Event, **SizeMode** Property.

ObjectTextFlow Property

Description: Informs about the way in which text flows around an embedded object.

Usage: TXTextControl.**ObjectTextFlow**

The property returns the following values:

Setting	Description
0	The object has been inserted 'as character' using the **ObjectInsertAsChar** method.
2	The object has been inserted as fixed object. The text stops at the top and continues at the bottom of the object.

3	The object has been inserted as fixed object. The text flows around the object and empty areas at the left and right side are filled.

Data Type: Integer.

Limitations: Read only, run time only.

See also: **ObjectInsertFixed** Method.

PageFormatChange Event

Description: Occurs when the page format settings have been changed.

Syntax: **PageFormatChange**( )

PageHeight Property

Description: Specifies the height of the page for the current document.

Usage: TXTextControl.**PageHeight** [= *value*]

Remarks: The height of the actual printed area is **PageHeight** minus **PageMarginB** minus **PageMarginT**. The maximum value depends on the capabilities of the selected printer and must not exceed 32767 twips.

If **PageHeight** is 0, the control's height is used instead. This setting can be used to place several controls without scrollbars on a page.

Data Type: Long.

See also: **PageWidth** Property, **PageMargin*x*** Properties, **PrintDevice** Property, **PrintPage** Method.

Example: See **PrintPage** Method example.

PageMarginB Property
PageMarginL Property
PageMarginR Property

PageMarginT Property

Description: Returns or sets the margins for the pages of the current document.

Usage: TXTextControl.**PageMarginB** [= *value*]
TXTextControl.**PageMarginL** [= *value*]
TXTextControl.**PageMarginR** [= *value*]
TXTextControl.**PageMarginT** [= *value*]

Remarks: The maximum values depend on the setting of the **PageWidth** and **PageHeight** properties.

Data Type: Long.

See also: **PageHeight** Property, **PageWidth** Property, **PrintDevice** Property, **PrintPage** Method.

Example: See **PrintPage** Method example.

PageWidth Property

Description: Specifies the width of the page for the current document.

Usage: TXTextControl.**PageWidth** [= *value*]

Remarks: The width of the actual printed area is **PageWidth** minus **PageMarginR** minus **PageMarginL**. The maximum value depends on the capabilities of the selected printer and must not exceed 32767 twips.

If **PageWidth** is 0, the control's width is used instead. This setting can be used to place several controls without scrollbars on a page.

Data Type: Long.

See also: **PageHeight** Property, **PageMargin*xx*** Properties, **PrintDevice** Property, **PrintPage** Method.

Example: See **PrintPage** Method example.

ParagraphChange Event

Description: Occurs when the character input position has been moved to another paragraph.

Syntax: **ParagraphChange**( )

ParagraphDialog Method

Description: Invokes the Text Control's built-in paragraph attributes dialog box and, after the user has closed the dialog box, specifies whether he has changed something.

Usage: TXTextControl.**ParagraphDialog**

Return Value: The method returns True when the user has changed one or more attibutes. The method returns False when the formatting remains unchanged.

Remarks: The changes, made in the dialog box, apply to the currently selected text.

Data Types: Return value: Boolean.

ParagraphFormatChange Event

Description: Occurs when the paragraph attributes of the selected paragraphs have been changed.

Syntax: **ParagraphFormatChange**( )

PosChange Event

Description: Occurs when the current character input position has been changed.

Syntax: **PosChange**( )

PrintColors Property

Description: Specifies whether text colors are printed as colors or in black.

Usage: TXTextControl.**PrinterColors** [= *boolean*]

The property's settings are:

Setting	Description
True	(Default) Text colors are printed.
False	All the text is printed in black.

Data Type: Boolean.

PrintDevice Property

Description: Sets the printer device context for TextContol's built-in printing function. The Windows operating enviroment manages devices like printers and screens with context handles.

Usage: TXTextControl.**PrintDevice** [= *DeviceContextHandle*]

Data Type: Long.

Limitations: Write only, run time only.

See also: **PageHeight** Property, **PageMargin*x*** Properties, **PageWidth** Property, **PrintPage** Method.

Example: See **PrintPage** Method.

PrintOffset Property

Description: Determines whether Text Control starts printing exactly at the top left corner of the page, or at the printer's physical printing offset.

Usage: TXTextControl.**PrintOffset** [= *boolean*]

The property's settings are:

Setting	Description
True	Adds the physical printing offset.
False	(Default) Do not add the physical printing offset.

Data Type: Boolean.

PrintPage Method

Description: Prints a page of text on the default printer. The number is specified through *PageNumber*. The first page has the number 1.

Usage: TXTextControl.**PrintPage** *PageNumber*

Return Value: This method has no return value.

Remarks: Prior to using this method the Text Control's output device must be selected using the **PrintDevice** property.

Data Types: *PageNumber* Integer

See also: **PageHeight** Property, **PageMargin*x*** Properties, **PageWidth** Property, **PrintDevice** Property.

Example: This Basic example shows how to print the contents of a Text Control on the default printer:

```
Sub mnuFile_Print_Click ()
   Dim wPages, No

   Printer.Print
   wPages = TXTextControl1.CurrentPages
   For No = 1 To wPages
      TXTextControl1.PrintDevice = Printer.hDC
      TXTextControl1.PrintPage No
      Printer.NewPage
   Next No
   Printer.EndDoc
End Sub
```

PrintZoom Property

Description: Specifies a zoom factor for printing. The value is specified as a percentage in the range of 10 to 400%. This property is independent of the current **ZoomFactor** setting.

Usage: TXTextControl.**PrintZoom** [= *value*]

Data Type: Integer.

See also: **ZoomFactor** Property.

Redo Method

Description: This method can be used to redo the last Text Control operation.

Usage: TXTextControl.**Redo**

Return Value: The method returns True if the redo operation was successful. Otherwise it returns False.

Data Types: Return value: Boolean

See also: **Undo** Method, **CanUndo** Property, **CanRedo** Property.

Refresh Method

Description: This method forces a complete repaint of a Text Control.

Usage: TXTextControl.**Refresh**

Return Value: This method has no return value.

ResetContents Method

Description: Deletes the complete contents of a Text Control including tables, objects, marked text fields and headers and footers.

Usage: TXTextControl.**ResetContents**

Return Value: The method returns True, if everything could be deleted, otherwise it returns False.

Data Types: Return value: Boolean

ResourceFile Property

Description: Returns or sets the file name of a resource library which Text Control loads to display resources like information strings, error messages and built-in dialog boxes. The file name must include a complete path. When a resource library is set, the value of the **Language** property is ignored. See the new chapter "*Overviews - Resources*" for more information.
To avoid compatibility errors, resource libraries should have a unique file name and should be placed in the same directory as the application's executable file.

Usage: TXTextControl.**ResourceFile** [= *string*]

Data Type: String

Limitations: Run time only.

See also: **Language** Property.

RTFSelText Property

Description: This property works much like the standard **SelText** property. The **SelStart** and **SelLength** properties can be used to specify a text selection which is to be copied to a string or inserted from a string. The difference between **SelText** and **RTFSelText** is that with the **SelText** property, text is stored without formatting information in the ANSI format, whilst **RTFSelText** uses Rich Text Format to preserve all of the formatting attributes.

Usage: TXTextControl.**RTFSelText** [= *string*]

Remarks: RTF (Rich Text Format) is one of the most common interchange formats for text documents. Most word processors available for Windows are able to read and write RTF files.

Data Type: String.

Limitations: Run time only.

See also: **RTFImport** Method.

RulerHandle Property

Description: Specifies the ruler control to be used with a Text Control.

Usage: TXTextControl.**RulerHandle** *[= Ruler.hWnd]*

Data Type: Handle

Limitations: Run time only.

See also: **StatusBarHandle** Property, **ButtonBarHandle** Property.

Save Method

Description: Saves data in a file with a specified format. All Unicode formats (6, 7 and 8) and the Microsoft Word format can only be used with the 32 bit version of Text Control.

Usage: TXTextControl.**Save** *FileName[, Offset[, Format[, CurSelection]]]*

The method's parameters are:

Parameter	Description
FileName	Specifies the file to save in.
Offset	Optional. Specifies the file position to where the data is to be written when the data of more than one Text Control are to be saved. When not specified or set to -1 the data is appended.
Format	Optional. Specifies a format identifier or the name of a user-developed filter. When this

parameter is not specified the data is saved in the internal Text Control format. The following format identifiers are possible:

1 - ANSI text	Text only in ANSI format (Windows compatible).
2 - TX text	Text only in ANSI format (Text Control compatible).
3 - TX	Internal Text Control format.
4 - HTML	HTML format (Hypertext Markup Language).
5 - RTF	RTF format (Rich Text Format).
6 - Unicode text	Text only in Unicode format (Windows compatible).
7 - TX text	Text only in Unicode format (Text Control compatible).
8 - TX	Internal Text Control format. Text is stored in Unicode.
9 - WORD	Microsoft Word format.
FilterFileName	Can be used with a user-developed filter.

CurSelection Optional. When set to true the current selection is saved. When omitted or set to false or when no selection exists all the control contents are saved.

Return Value: The method returns the position in the file behind the saved data. It is zero if the data could not be saved.

Data Types:

FileName:	String
Offset:	Long
Format:	Integer or String

CurSelection: Boolean
Return value: Long

See also: **Load** Method.

SaveToMemory Method

Description: Returns a byte array containing text data in a specified format. This method works in the same way as the **Save** method.

Usage: TXTextControl.**SaveToMemory***[Format[, CurSelection]]*

The method's parameters are:

Parameter	Description
Format	Optional. See the *Format* parameter of the **Save** method for a description.
CurSelection	Optional. See the *CurSelection* parameter of the **Save** method for a description.

Return Value: The method returns a one-dimensional array of bytes, containing the saved data.

Data Types: *Format*: Integer or String
CurSelection: Boolean
Return value: One-dimensional Byte Array

See also: **Save** Method, **LoadFromMemory** Method

Example: This Basic example copies the currently selected text from the first Text Control and inserts it at the current input position of a second Text Control:

```
Dim data() As Byte

data = TXTextControl1.SaveToMemory(3, True)

TXTextControl2.LoadFromMemory data, 3, True
```

ScrollBars Property

Description: Returns or sets a value indicating whether a Text Control has horizontal or vertical scroll bars. Scroll bars are automatically hidden when the formatting area is smaller than the control's visible area and vice versa. Therefore this property has only effect if the **PageWidth** and/or the **PageHeight** properties have been set to non-zero. See "*Overviews - Text Formatting and Views*" for more information.

Usage: TXTextControl.**ScrollBars** [= *value*]

The property's settings are:

Setting	Description
0	(Default) The Text Control has no scroll bars.
1	The Text Control has a horizontal scroll bar when the page width is larger than the control's width.
2	The Text Control has a vertical scroll bar when the page height is larger than the control's height.
3	The Text Control has both scroll bars.

Data Type: Integer.

See also: **PageWidth** Property, **PageHeight** Property

ScrollPosX Property

Description: Specifies the position of the horizontal scroll bar in twips.

Usage: TXTextControl.**ScrollPosX** [= *value*]

Data Type: Long.

Limitations: Run time only.

See also: **ScrollPosY** Property, **HScroll** Event, **VScroll** Event.

ScrollPosY Property

Description: Specifies the position of the vertical scroll bar in twips.

Usage: TXTextControl.**ScrollPosY** [= *value*]

Data Type: Long.

Limitations: Run time only.

See also: **ScrollPosX** Property, **HScroll** Event, **VScroll** Event.

SelLength Property

Description: Returns or sets the number of characters selected.

Usage: TXTextControl.**SelLength** [= *value*]

Remarks: The valid range of settings is 0 to text length, the total number of characters a Text Control contains.

Data Type: Long.

Limitations: Run time only.

See also: **SelStart** Property, **SelText** Property.

SelStart Property

Description: Returns or sets the starting point of text selected or indicates the position of the insertion point if no text is selected. The first text position has a value of 0.

Usage: TXTextControl.**SelStart** [= *value*]

Remarks: Setting the property value greater than the text length limits it to the existing text length.

Data Type: Long.

Limitations: Run time only.

See also: **SelLength** Property, **SelText** Property.

SelText Property

Description: Returns or sets the string containing the currently selected text. It is a zero-length string if no characters are selected. This property can be used in conjunction with the **SelStart** and **SelLength** properties for tasks such as selecting substrings or clearing text. In conjunction with the clipboard these properties are useful for copy, cut, and paste operations.

Usage: TXTextControl.**SelText** [= *value*]

Remarks: Setting this property to a new value sets **SelLength** to 0 and replaces the selected text with the new string.

Data Type: String.

Limitations: Run time only.

See also: **SelLength** Property, **SelStart** Property.

Size Event

Description: Occurs when a Text Control has been resized with the mouse while depressing the ALT key.

Syntax: **Size**( )

See also: **Move** Event, **SizeMode** Property.

SizeMode Property

Description: Specifies whether the Text Control can be moved or resized at run time. If the Moveable option is selected, the control can be moved by depressing the ALT key and then dragging the control with the mouse. If the Sizeable option is selected and the ALT key is depressed, the borders of the control can be dragged.

Usage: TXTextControl.**SizeMode** [= *value*]

The property's settings are:

Setting	Description
0 - Fixed	(Default) The Text Control cannot be moved or sized.
1 - Moveable	The Text Control can be moved.
2 - Sizeable	The Text Control can be sized.
3 - Move and Sizeable	The Text Control can be moved and sized.

Data Type: Integer.

See also: **Move** Event, **Size** Event.

StatusBarHandle Property

Description: Specifies the status bar control to be used with a Text Control.

Usage: TXTextControl.**StatusBarHandle** *[= TXStatusBar.hWnd]*

Data Type: Handle

Limitations: Run time only.

See also: **ButtonBarHandle** Property, **RulerHandle** Property.

TabCurrent Property

Description: Specifies the current tab number for the properties **TabPos** and **TabType**.

Usage: TXTextControl.**TabCurrent** [= *value*]

Remarks: Text Control supports up to 14 tabs for each paragraph. Valid values for this property are 1 to 14.

Data Type: Integer.

See also: **TabPos** Property, **TabType** Property.

Example: This Basic example moves the first tab to a new position 1 inch from the left border and changes it to a decimal tab:

```
TXTextControl1.TabCurrent = 1
```

```
TXTextControl1.TabType = 4
TXTextControl1.TabPos = 1440
```

This Basic example changes all the tabs to be right-aligned at 1/2 inch gradations:

```
' Delete all tabs
for n=14 to 1 step -1
   TXTextControl1.TabCurrent = n
   TXTextControl1.TabPos = 0
next n

' Create new tabs
for n=1 to 14
   TXTextControl1.TabCurrent = n
   TXTextControl1.TabPos = n*720
   if (TXTextControl1.TabPos > 0) then
TXTextControl1.TabType = 2
next n
```

Text Control sorts the tabs in ascending order whenever you change the position of a tab, so a tab's number can change when it it moved. In this case, the **TabCurrent** Property is updated to reflect the change.

Tabs outside of the paragraph are automatically set to zero.

TabKey Property

Description: Determines if the keyboard's tab key moves the input focus to the next control or to insert tabulators in the Text Control's text.

Usage: TXTextControl.**TabKey** [= *boolean*]

The property's settings are:

Setting	Description
True	(Default) Inserts a tabulator in the Text Control's text.
False	Moves the current input focus to the next control.

Data Type: Boolean.

TableAtInputPos Property

Description: Returns the table identifier of the table containing the input position. Zero is returned when the input position is not inside a table or when more than one table cell is selected.

Usage: TXTextControl.**TableAtInputPos**

Data Type: Integer.

Limitations: Read only, run time only.

See also: **TableColAtInputPos** Property, **TableRowAtInputPos** Property

TableAttrDialog Method

Description: This method invokes the Text Control's built-in table-attributes dialog box and, after the user has closed the dialog box, specifies whether he has changed something.

Usage: TXTextControl.**TableAttrDialog**

Return Value: The method returns True when the user has changed one or more attibutes. The method returns False when the formatting remains unchanged.

Remarks: The changes, made in the dialog box, apply to the currently selected text.

Data Types: Return value: Boolean.

TableCanChangeAttr Property

Description: This property provides information about whether the attributes of all the selected table cells can be altered. It returns False when the selection is not completely within a single table. Otherwise it returns True.

Usage: TXTextControl.**TableCanChangeAttr**

The property's settings are:

Setting	Description
True	Table attributes can be altered.
False	Table attributes cannot be altered.

Data Type: Boolean.

Limitations: Read only, run time only.

See also: **TableAttrDialog** Method.

TableCanDeleteLines Property

Description: This property provides information about whether table lines can be deleted. It returns False when no table line is selected or when the current input position is outside a table. Otherwise it returns True.

Usage: TXTextControl.**TableCanDeleteLines**

The property's settings are:

Setting	Description
True	Table lines can be deleted.
False	Table lines cannot be deleted.

Data Type: Boolean.

Limitations: Read only, run time only.

See also: **TableDeleteLines** Method.

TableCanInsert Property

Description: This property provides information about whether a table can be inserted. It returns False when a selection exists or the current input position is inside a table. Otherwise it returns True.

Usage: TXTextControl.**TableCanInsert**

The property's settings are:

Setting	Description
True	A new table can be inserted.
False	A new table cannot be inserted.

Data Type: Boolean.

Limitations: Read only, run time only.

See also: **TableInsert** Method.

TableCellAttribute Property

Description: Returns or sets attributes of a table cell.

Usage: TXTextControl.TableCellAttribute(*TableId,Row,Column,Attribute*) [= *value*]

The property's parameters are:

Parameter	Description
TableId	Specifies a table. It is the same identifier set with the **TableInsert** method.
Row	Specifies a row number which identifies a certain cell in the table. The first row has the number one. This parameter set to zero means a complete column.
Column	Specifies a column number which identifies a certain cell in the table. The first column has the number one. This parameter set to zero means a complete row.
Attribute	Specifies the type of attribute. It can be anyone of the values listed in Constants.

Constants: Valid constants for *Attribute* are:

Attribute	Description
txTableCellHorizontalPos	The property value is the horizontal position of a table cell.

txTableCellHorizontalExt	The property value is the horizontal extension of a table cell.
txTableCellBorderWidth	The property value is the border width of a table cell.
txTableCellTextGap	The property value is the width of the gap between a cell's border and its text.
txTableCellBackColor	The property value is the table cell's background color.
txTableCellLeftBorderWidth	The property value is the left border width of a table cell.
txTableCellTopBorderWidth	The property value is the top border width of a table cell.
txTableCellRightBorderWidth	The property value is the right border width of a table cell.
txTableCellBottomBorderWidth	The property value is the bottom border width of a table cell.
txTableCellLeftTextGap	The property value is the width of the gap between a cell's left border and its text.
txTableCellTopTextGap	The property value is the width of the gap between a cell's top border and its text.
txTableCellRightTextGap	The property value is the width of the gap between a cell's right border and its text.
txTableCellBottomTextGap	The property value is the width of the gap between a cell's bottom border and its text.

Remarks: When the row and column parameters are both set to zero the attributes of a complete table can be manipulated.
When more than one table cell is specified this property returns Null

(Visual Basic) or is set to empty (C++) if the cells are differently formatted.

Data Types:

TableId	Integer
Row	Integer
Column	Integer
Attribute	Integer
Property value:	Variant

Limitations: Run time only.

See also: **TableInsert** Method, **TableCellText** Property

Example: This Basic example sets a red background color for the leftmost column of a table:

```
TXTextControl1.TableCellAttribute(id, 0, 1, 4) = RGB(255, 0
,0)
```

TableCellLength Property

Description: Returns the number of characters in a table cell.

Usage: TXTextControl.**TableCellLength**(*TableId*,*Row*,*Column*)

The property's parameters are:

Parameter	Description
TableId	Specifies a table. It is the same identifier set with the **TableInsert** method.
Row, *Column*	Specify a row and column number which identifies a certain cell in the table. The first has the number 1, 1.

Data Types:

TableId	Integer
Row	Integer
Column	Integer
Property value:	Long

Limitations: Read only, run time only.

TableCellStart Property

Description: Returns the character index (one-based) of the first character in a table cell.

Usage: TXTextControl.**TableCellStart**(*TableId,Row,Column*)

The property's parameters are:

Parameter	Description
TableId	Specifies a table. It is the same identifier set with the **TableInsert** method.
Row, Column	Specify a row and column number which identifies a certain cell in the table. The first cell has the number 1, 1.

Data Types:

TableId	Integer
Row	Integer
Column	Integer
Property value:	Long

Limitations: Read only, run time only.

TableCellText Property

Description: Returns or sets the text of a table cell.

Usage: TXTextControl.**TableCellText**(*TableId,Row,Column*) [= *string*]

The property's parameters are:

Parameter	Description
TableId	Specifies a table. It is the same identifier set with the **TableInsert** method.
Row, Column	Specify a row and column number which identifies a certain cell in the table. The first cell has the number 1, 1.

Data Types:

TableId	Integer
Row	Integer

	Column	Integer
	Property value:	String

Limitations: Run time only.

See also: **TableInsert** Method, **TableRows** Property, **TableColumns** Property

TableColAtInputPos Property

Description: Returns the number of the current input column in a table. It is zero when the input position is not inside a table or when more than one table cell is selected.

Usage: TXTextControl.**TableColAtInputPos**

Data Type: Integer.

Limitations: Read only, run time only.

See also: **TableAtInputPos** Property, **TableRowAtInputPos** Property

TableColumns Property

Description: Informs about the number of columns a specified table contains.

Usage: TXTextControl.**TableColumns**(*TableId*)

The property's parameters are:

Parameter	**Description**
TableId	Specifies a table. It is the same identifier set with the **TableInsert** method.

Data Types:	*TableId*	Integer
	Property value:	Integer

Limitations: Read only, run time only.

See also: **TableInsert** Method, **TableRows** Property

TableCreated Event

Description: Occurs after a new table has been created as a result of a text insertion via the clipboard. It does not occur when the table is inserted with the **TableInsert** method or when a previously saved document is reloaded.

Syntax: **TableCreated**(*TableId*, *NewTableId*)

The event procedure's parameters are:

Parameter	Description
TableId	Is the number of the created table. This number can be changed through setting the *NewTableId* parameter.
NewTableId	Is a new table identifier for the created table. It must be in the range of 10 to 32,767.

Data Types:

TableId	Integer
NewTableId	Integer

See also: **TableDeleted** Event, **TableInsert** Method.

TableDeleted Event

Description: Occurs after a table has been deleted.

Syntax: **TableDeleted**(*TableId*)

The event procedure's parameters are:

Parameter	Description
TableId	Is the the identifier of the deleted table.

Data Types:

TableId	Integer

See also: **TableCreated** Event.

TableDeleteLines Method

Description: This method deletes the currently selected table lines or the table line at the current input position.

Usage: TXTextControl.**TableDeleteLines**

Return Value: The method returns True if table lines have been deleted. Otherwise it returns False.

Data Types: Return value: Boolean

See also: **TableInsert** Method, **TableCanDeleteLines** Property.

TableGridLines Property

Description: This property determines whether or not grid lines in tables are visible.

Usage: TXTextControl.**TableGridLines** [= *boolean*]

Remarks: The property's settings are:

Setting	Description
True	(Default) Grid lines in tables are visible.
False	Grid lines in tables are invisible.

Data Type: Boolean.

TableInsert Method

Description: This method inserts a new table in the text.

Usage: TXTextControl.**TableInsert** *Rows, Columns, TextPos [, TableId]*

The method's parameters are:

Parameter	Description
Rows	Specifies the number of rows.
Columns	Specifies the number of columns.

TextPos	Specifies the text position where the new table is to be inserted. It is inserted at the current input position when this parameter is set to -1.
TableId	Optional. Specifies a table identifier. This identifier can be used to access or to alter the table's text and attributes. It must be in the range of 10 to 32,767.

Return Value: The method returns one of the following values:

Value	Description
0	An error has occurred or the table could not be inserted. Tables cannot be inserted inside existing tables or when a section of text has been selected.
-1	The new table has been inserted at the top or at the bottom of an existing table and has been combined with this table.
otherwise	The table's identifier. This is the same value as specified with the *TableId* parameter or an internal identifier selected by Text Control when the optional *TableId* parameter has been omitted.

Data Types:

Rows	Integer
Columns	Integer
TextPos	Long
TableId	Integer
Return value:	Integer

See also: **TableCanInsert** Property, **TableDeleteLines** Method.

TableNext Method

Description: This method returns a enumaration number of the table that follows the specified table in the Text Control's current text. It can be used to enumerate all tables. In a list of linked Text Controls the search is

performed in all controls. The method uses enumeration numbers instead of table identifiers because table identifiers are not unique. The corresponding table identifer is retrieved by the *TableId* parameter.

Usage: TXTextControl.**TableNext** *EnumerationNumber*, *TableId*

The method's parameters are:

Parameter	Description
EnumerationNumber	Specifies a enumeration number. The method returns the enumeration number of the table that follows the table with this number. If this parameter is zero the first table's enumeration number is returned.
TableId	Text Control copies the table's identifier to this variable. This is the same value set with the **TableInsert** method.

Return Value: Specifies the enumeration number of the next table. It can be used for the next **TableNext** call. The return value is zero when the last table has been reached or when the specified enumeration number was invalid.

Data Types:

EnumerationNumber	Integer
TableId	Integer
Return value:	Integer

TableRowAtInputPos Property

Description: Returns the number of the current input row in a table. It is zero when the input position is not inside a table or when more than one table cell is selected.

Usage: TXTextControl.**TableRowAtInputPos**

Data Type: Integer.

Limitations: Read only, run time only.

See also: **TableAtInputPos** Property, **TableColAtInputPos** Property

TableRows Property

Description: Informs about the number of rows a specified table contains.

Usage: TXTextControl.**TableRows**(*TableId*)

The property's parameters are:

Parameter	Description
TableId	Specifies a table. It is the same identifier set with the **TableInsert** method.

Data Types: *TableId* Integer
Property value: Integer

Limitations: Read only, run time only.

See also: **TableInsert** Method, **TableColumns** Property

TabPos Property

Description: Determines the position (in twips) of a certain tab. The tab number must have previously been determined with the **TabCurrent** property.

Usage: TXTextControl.**TabKey** [= *value*]

Data Type: Long.

See also: **TabCurrent** Property, **TabType** Property.

TabType Property

Description: Determines the type of a certain tab. The tab number must have previously been determined with the **TabCurrent** property.

Usage: TXTextControl.**TabType** [= *value*]

The property's settings are:

Setting	Description
1	Left tab.

2	Right tab.
3	Centered tab.
4	Decimal tab.
5	Right tab at the right most text position. For this type any position set with the **TabPos** property is ignored.

Data Type: Integer.

See also: **TabCurrent** Property, **TabPos** Property.

Text Property

Description: Returns or sets the complete text of a Text Control.

Usage: TXTextControl.**Text** [= *string*]

Data Type: String.

TextBkColor Property

Description: Returns or sets the background color for selected text. Text Control uses the Microsoft Windows operating enviroment red-green-blue (RGB) color scheme.

Usage: TXTextControl.**TextBkColor** [= *value*]

Remarks: The **TextBkColor** property applies only to the currently selected text. The **BackColor** property can be used to set the window background color.
The valid range for a RGB color is 0 to &HFFFFFF. The high byte of a number in this range equals 0; the lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF).

Data Type: Long.

See also: **FormatSelection** Property, **BackColor** Property.

Undo Method

Description: The **Undo** method can be used to undo the last Text Control operation.

Usage: TXTextControl.**Undo**

Return Value: The method returns True if the undo operation was successful. Otherwise it returns False.

Data Types: Return value: Boolean

See also: **Redo** Method, **CanUndo** Property, **CanRedo** Property.

VExpand Event

Description: Occurs when the control has changed its window size vertically. This event can only occur if the **AutoExpand** property is set to True.

Syntax: **VExpand**( )

See also: **AutoExpand** Property, **HExpand** Event.

ViewMode Property

Description: Returns or sets the mode in which Text Control displays the document pages. This property has only effect if the **PageWidth** and/or **PageHeight** properties have been set to non-zero. See "*Overviews - Text Formatting and Views*" for more information.

Usage: TXTextControl.**ViewMode** [= *value*]

The property's settings are:

Setting	Description
0 - Normal view	(Default) Do not display page borders, margins and gaps.
1 - Page view	Display the document's pages with page margins and show the page number in the status bar.

	2 - Ext. page view	Shows the document's pages centered and displays three-dimensional borders.

Data Type: Integer.

See also: **PageWidth** Property, **PageHeight** Property.

VScroll Event

Description: Occurs when the vertical scroll position has been changed.

Syntax: **VScroll**( )

See also: **HScroll** Event.

VTSpellCheck Method

Description: Starts the spellchecker. This method is only available if the VT-Speller tool from VisualTools has been installed. VT Speller is not part of the Text Control package.

Usage: TXTextControl.**VTSpellCheck**

Return Value: The method returns True if the spellchecker could be started, otherwise it returns False.

Data Types: Return value: Boolean

See also: **VTSpellDictionary** Property.

VTSpellDictionary Property

Description: Determines the file name of the dictionary which is used by VT-Speller.Text Control uses this property only if the VT-Speller tool from VisualTools has been installed. VT Speller is not part of the Text Control package.

Usage: TXTextControl.**VTSpellDictionary** [= *string*]

Data Type: String.

See also: **VTSpellCheck** Method.

Zoomed Event

Description: Occurs when the Text Control has been zoomed.

Syntax: **Zoomed**( )

See also: **ZoomFactor** Property.

ZoomFactor Property

Description: Specifies the zoom factor for a Text Control. The value is specified as a percentage in the range of 10..400%.

Usage: TXTextControl.ZoomFactor [= *value*]

Data Type: Integer.

See also: **PrintZoom** Property.

Obsolete Properties, Events, and Methods

The following is a list of obsolete properties, methods and events . These are still provided for compatibility with earlier versions of Text Control. Newly developed applications should use the appropriate newer properties, methods or events.

Property/Method/Event	Description
EnableHyperlinks Property	Enables special actions for pieces of text that work as hyperlinks. Text parts that function as hyperlinks are now automatically converted to marked text fields, when the **LoadSaveAttribute**(**txEnableLinks**) property has been set to True before a document is loaded.
LoadSaveAttribute(53) Property	Enables automatic jumps when the user clicks on a hypertext link. Can now be realized with the new **FieldGoto** method.
RTFExport Method	Has been replaced with the **Save** method.
RTFImport Method	Has been replaced with the **Load** method.
TextExport Method	Has been replaced with the **Save** method.
TextImport Method	Has been replaced with the **Load** method.
ViewClicked Event	Has been replaced with the new **FieldLinkClicked** event.
ViewImagePath Property	Has been replaced with the new **LoadSaveAttribute**(**txAbsPath**) property.

ViewNextHighlight Method	Scrolls to the next highlight. Can be realized with the new **FieldGoto** method.
ViewSection Property	Jumps to a specified text position. Can be realized with the new **FieldGoto** method.
ViewWordDblClicked Event	Occurs when the mouse is double-clicked over text in Viewer mode.

EnableHyperlinks Property

Description: This property must be set to True to enable special actions for pieces of text that function as hyperlinks. After loading a document that contains hyperlinks the user receives **ViewClicked** and **ViewWordDblClicked** events and can use the **ViewSection** and **ViewNextHighlight** properties and methods. With enabled hyperlinks a Text Control cannot be edited.

Usage: TXTextControl.**EnableHyperlinks** [= *boolean*]

The property's settings are:

Setting	Description
True	Hyperlinks are enabled. The **Viewxx** events and methods are available.
False	(Default) Hyperlinks are disabled. The **Viewxx** events and methods are not available.

Data Type: Boolean.

See also: **LoadSaveAttribute** Property.

RTFExport Method

Description: Writes the contents of a Text Control to a file with the specified name using the Rich Text Format.

Usage: TXTextControl.**RTFExport** *Filename*

Return Value: The method returns True when the data has been written to the file. Otherwise it returns False.

Remarks: RTF (Rich Text Format) is one of the most common interchange formats for text documents. Most word processors available for Windows are able to read and write RTF files.

Data Types:

FileName	String
Return value:	Boolean

See also: **RTFImport** Method.

RTFImport Method

Description: Loads the contents of an RTF file with the specified name into a Text Control.

Usage: TXTextControl.**RTFImport** *Filename[, extended]*

The method's parameters are:

Parameter	Description
Filename	Is the name of the RTF file that is to be loaded.
Extended	Optional. If this parameter is missing or zero then the text is inserted at the current caret position. If this parameter has a value of 1 then Text Control supports a special Viewer mode. In this mode, additional hypertext information is imported from the RTF file. See TX Info Artist manual for details.

Return Value: The method returns True when the data could be imported. Otherwise it returns False.

Remarks: RTF (Rich Text Format) is one of the most common interchange formats for text documents. Most word processors available for Windows are able to read and write RTF files.

Data Types: *Filename*: String
Extended: Long
Return value: Boolean

See also: **RTFExport** Method, **ViewImagePath** Property.

TextExport Method

Description: Writes the selected text to a file in ANSI format.

Usage: TXTextControl.**TextExport** *FileName*

The method's parameters are:

Parameter	Description
FileName	Is the name of the file, Text Control uses for saving. When the file does not exist, a new file with this name is created. When the file exists, Text Control overwrites its current contents.

Return Value: The method returns True when the text has been written to the file. Otherwise it returns False.

Data Types: *FileName* String
Return value: Boolean

See also: **TextImport** Method, **RTFImport** Method, **Save** Method.

TextImport Method

Description: Loads text in ANSI format and inserts it at the current caret position.

Usage: TXTextControl.**TextImport** *FileName*

The method's parameters are:

Parameter	Description
FileName	Is the name of the file, Text Control uses for loading.

Return Value: The method returns True when the text could be imported. Otherwise it returns False.

Data Types:

FileName	String
Return value	Boolean

See also: **TextExport** Method, **RTFExport** Method, **Load** Method.

ViewClicked Event

Description: Occurs when a marked text field for which hyperlink data has been stored, is clicked on. It occurs only if the **EnableHyperlinks** property is set to True.

Syntax: **ViewClicked**(*FieldType*, *FieldContents*)

The event procedure's parameters are:

Parameter	Description
FieldType	Specifies the type of a field as an identifier. It can be one of the following values: 0 - 19 Specifies fields that represent links to text positions within the same document (RTF only). 20 - 39 Specifies buttons that represent link positions (RTF only). 100 Specifies a field that is a link to a position in the same document (HTML only). 101 Specifies a field that is a link to an external position (HTML only).
FieldContents	Is a string that represents the contents of the field. The specification depends on the field's type and on the type of the document.

Remarks: This event is sent before a **FieldClicked** event is sent.

Data Types:

FieldType	Integer
FieldContents	String

See also: **EnableHyperlinks** Property.

ViewImagePath Property

Description: When importing text data files which contain references to image files, the **ViewImagePath** property can be used to specify a different path for the images.

Usage: TXTextControl.**ViewImagePath** [= *string*]

Data Type: String.

Limitations: Run time only.

See also: **RTFImport** Method.

ViewNextHighlight Method

Description: Scrolls to the next highlight.

Usage: TXTextControl.**ViewNextHighlight**

Return Value: The method returns False if the last highlight has been reached. Otherwise it returns True.

Remarks: The use of this method is only valid if the **RTFImport** method was previously used with the *extended* parameter set to 1. See TX Info Artist description for details.

Data Types: Return value: Boolean

See also: **RTFImport** Method.

ViewSection Property

Description: Jumps to a specified text position. See TX Info Artist description for details.

Usage: TXTextControl.ViewSection [= *SectionNumber*]

Remarks: The use of this property is only valid if the **RTFImport** method was used before with the *extended* parameter set to 1.

Data Type: Integer.

Limitations: Write only, run time only.

See also: **RTFImport** Method.

ViewWordDblClicked Event

Description: Occurs when a word is double-clicked with the mouse. It occurs only if the **EnableHyperlinks** property is set to True.

Syntax: **ViewWordDblClicked**(*SelectedText*)

The event procedure's parameters are:

Parameter	Description
SelectedText	Is the word that has been double-clicked.

Remarks: This event is sent before a **DblClick** event is sent.

Data Type: *SelectedText* String

See also: **RTFImport** Method, **DblClick** Event.

Button Bar Control Properties, Events, and Methods

All of the properties, methods and events for the button bar are listed in alphabetical order in the following table. A detailed description can be found in the following section.

Properties	Events
Appearance	MouseDown
BorderStyle	MouseMove
Enabled	MouseUp
hWnd	
Language	
ResourceFile	
Style	

BorderStyle Property
Enabled Property
hWnd Property
Language Property
MouseDown Event
MouseMove Event
MouseUp Event
ResourceFile Property

All of these properties and events work in the same way as for a Text Control. See the appropriate section prior in this manual.

Appearance Property

Description: Returns or sets the paint style of a Button Bar.

Usage: TXButtonBar.**Appearance** [= *value*]

The property's settings are:

Setting	Description
0 - txFlat	Flat. Paints the Button Bar without visual effects.
1 - tx3D	(Default). 3D. Paints the Button Bar with three-dimensional effects.

Data Type: Integer.

Style Property

Description: Returns or sets the paint style of a Button Bar's buttons.

Usage: TXButtonBar.**Style** [= *value*]

Remarks: The **Style** property settings are:

Setting	Description
0 - txFlat	Flat. Paints the Button Bar's buttons without visual effects.
1 - tx3D	(Default) 3D. Paints the Button Bar's buttons with three-dimensional effects.

Data Type: Integer.

Status Bar Control Properties, Events, and Methods

All of the properties, methods and events for the status bar are listed in alphabetical order in the following table. A detailed description can be found in the following section.

Properties	Events
BorderStyle	MouseDown
Enabled	MouseMove
Font	MouseUp
FontBold	
FontItalic	
FontName	
FontSize	
FontStrikethru	
FontUnderline	
hWnd	
Language	
PageMode	
ResourceFile	
Text	
TextColumn	
TextLine	
TextPage	

BorderStyle Property
Enabled Property
hWnd Property
Language Property
MouseDown Event
MouseMove Event
MouseUp Event
ResourceFile Property

All of these properties and events work in the same way as for a Text Control. See the appropriate section prior in this manual.

FontBold Property
FontItalic Property
FontStrikethru Property
FontUnderline Property

Description: Returns or sets font styles in the following formats: **Bold**, *Italic*, ~~Strikethru~~, and Underline.

Usage: TXStatusBar.**FontBold** [= *boolean*]
TXStatusBar.**FontItalic** [= *boolean*]
TXStatusBar.**FontStrikethru** [= *boolean*]
TXStatusBar.**FontUnderline** [= *boolean*]

The properties' settings are:

Setting	Description
True	The characters are formatted with the specified style.

	False	The characters are not formatted with the specified style.

Data Type: Boolean.

See also: **FontName** Property, **FontSize** Property.

FontName Property

Description: Returns or sets the font used to display text.

Usage: TXStatusBar.**FontName** [= *string*]

See also: **FontSize** Property.

FontSize Property

Description: Returns or sets a value that specifies the size of the font used to display text.

Usage: TXStatusBar.**FontSize** [= *value*]

Data Type: Integer.

See also: **FontName** Property.

PageMode Property

Description: Returns or sets the status of the StatusBar's 'Page' field.

Usage: TXStatusBar.**PageMode** [= *value*]

The property's settings are:

Setting	Description
0	The 'Page' field is hidden.
1	The 'Page' field is always shown.
2	The 'Page' field is only shown if the connected Text-Contol's **ViewMode** property has been set to 'Page View' or 'Extended Page View' or

if several Text Controls are linked with the **NextWindow** property.

Data Type: Integer.

Text Property

Description: Returns or sets the info text a Status Bar Control displays.

Usage: TXStatusBar.**Text** [= *string*]

Data Type: String.

TextColumn Property
TextLine Property
TextPage Property

Description: Returns or sets the texts which appear in the 'Column', 'Line' and 'Page' fields of the Status Bar. By default "Col", "Line" and "Page" is displayed.

Usage: TXStatusBar.**TextColumn** [= *string*]
TXStatusBar.**TextLine** [= *string*]
TXStatusBar.**TextPage** [= *string*]

Data Type: String.

Ruler Control Properties, Events, and Methods

All of the properties, methods and events for the ruler are listed in alphabetical order in the following table. A detailed description can be found in the following section.

Properties	Events
BorderStyle	MouseDown
Enabled	MouseMove
hWnd	MouseUp
Language	
ResourceFile	
ScaleUnits	

BorderStyle Property
Enabled Property
hWnd Property
Language Property
MouseDown Event
MouseMove Event
MouseUp Event
ResourceFile Property

All of these properties and events work in the same way as for a Text Control. See the appropriate section prior in this manual.

ScaleUnits Property

Description: Returns or sets the scale units for the Ruler.

Usage: TXRuler.**ScaleUnits** [= *value*]

The property's settings are:

Setting	Description
0	mm
1	cm
2	inch

Data Type: Integer.

PageRuler Properties, Events, and Methods

All of the Properties, Events, and Methods for the Page Ruler are listed in alphabetical order in the following table. Properties and Events that apply only to this control are marked with an asterisk (*) and documented in the following section:

Properties

Align
BackColor
ClipControls
CtlName
DragIcon
DragMode
Enabled
Height
HelpContextID
hWnd
Index
Left
MousePointer
*OriginX
*OriginY
Parent
Tag
Top
*Units
Visible
Width
*ZoomFactor

Events

Click
DblClick
DragDrop
DragOver
MouseDown
MouseMove
MouseUp

Methods

Move
Refresh
SetFocus
ZOrder

OriginX and OriginY Properties

Description: Specify the ruler origin, i.e. the distance between the point where the ruler displays its 0 coordinate and the top left corner of the ruler window. Not available at design time. The measurement depends on the selected scale mode.

Usage: [form.]PgRuler.OriginX = *origin*

Data Type: Long.

Units Property

Description: Specifies if the ruler is to display its scale in inches or centimeters.

Usage: [form.]PgRuler.Units = *units*

Remarks: The settings are:

Setting	Description
0	cm
1	inch

Data Type: Integer.

ZoomFactor Property

Description: Specifies the zoom factor as a percentage.

Usage: [form.]PgRuler.ZoomFactor = *zoom factor*

Data Type: Integer.

Appendix A: Mouse and Keyboard Assignment

Mouse Assignment

Mouse Action	Reaction of Text Control
Click	Moves cursor to point of click or selects an image.
Shift+Click	Extends the selection to the point of click.
Double-click	Selects the word that is clicked on or opens a modal dialog box to select an image alignment.
Drag	Selects text from point of button down to point where button is released.
Double-click and drag	Extends the selection from word to word.
Triple-click and drag	Extends the selection from row to row.
PgUp/PgDown	Scrolls the text up or down one client area height minus the height of one line of text. Active only if a vertical scrollbar exists.

Moving the caret while SHIFT is pressed extends the current selection to the new caret position.

Keyboard Assignment

Key type	Reaction of Text Control
HOME	Moves the caret to the beginning of the line.
END	Moves the caret to the end of the line.
(Left Arrow)	Moves the caret one character to the left.
(Right Arrow)	Moves the caret one character to the right.
(Up Arrow)	Moves the caret one line up.
(Down Arrow)	Moves the caret one line down.
CTRL+(Left Arrow)	Moves the caret to the beginning of the current word.

CTRL+(Right Arrow)	Moves the caret to the beginning of the next word.
CTRL+HOME	Moves the caret to start of text.
CTRL+END	Moves the caret to end of text.
CTRL+ENTER	Inserts a new page.
SHIFT+ENTER	Creates a line feed.
CTRL+(-)	Inserts an end-of-line hyphen.
DEL	Deletes selected text.
SHIFT+DEL	Copies selected text to the Clipboard and deletes the selection.
CTRL+INS	Copies selected text to the clipboard.
SHIFT+INS	Inserts text from the clipboard.
CTRL+SHIFT+(Spacebar)	Inserts a non-breaking space.
CTRL+(Backspace)	Deletes the previous word.

Moving the caret while SHIFT is pressed extends the current selection to the new caret position.

Index

H

I

K

L

M

N

O

P

R

S

T

V

Z